Chapter 1

INTRODUCTION

1.1 WHAT IS AN ALGORITHM?

The word algorithm comes from the name of a Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who wrote a textbook on mathematics. This word has taken on a special significance in computer science, where "algorithm" has come to refer to a method that can be used by a computer for the solution of a problem. This is what makes algorithm different from words such as process, technique, or method.

Definition 1.1 [Algorithm]: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input.** Zero or more quantities are externally supplied.

2. **Output.** At least one quantity is produced.

3. **Definiteness.** Each instruction is clear and unambiguous.

4. **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible. □

An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include.

Criteria 1 and 2 require that an algorithm produce one or more *outputs* and have zero or more *inputs* that are externally supplied. According to criterion 3, each operation must be *definite*, meaning that it must be perfectly clear what should be done. Directions such as "add 6 or 7 to x" or "compute 5/0" are not permitted because it is not clear which of the two possibilities should be done or what the result is.

The fourth criterion for algorithms we assume in this book is that they *terminate* after a finite number of operations. A related consideration is that the time for termination should be reasonably short. For example, an algorithm could be devised that decides whether any given position in the game of chess is a winning position. The algorithm works by examining all possible moves and countermoves that could be made from the starting position. The difficulty with this algorithm is that even using the most modern computers, it may take billions of years to make the decision. We must be very concerned with analyzing the efficiency of each of our algorithms.

Criterion 5 requires that each operation be *effective*; each step must be such that it can, at least in principle, be done by a person using pencil and paper in a finite amount of time. Performing arithmetic on integers is an example of an effective operation, but arithmetic with real numbers is not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violate the effectiveness property.

Algorithms that are definite and effective are also called *computational procedures*. One important example of computational procedures is the operating system of a digital computer. This procedure is designed to control the execution of jobs, in such a way that when no jobs are available, it does not terminate but continues in a waiting state until a new job is entered. Though computational procedures include important examples such as this one, we restrict our study to computational procedures that always terminate.

To help us achieve the criterion of definiteness, algorithms are written in a programming language. Such languages are designed so that each legitimate sentence has a unique meaning. A *program* is the expression of an algorithm in a programming language. Sometimes words such as procedure, function, and subroutine are used synonymously for program. Most readers of this book have probably already programmed and run some algorithms on a computer. This is desirable because before you study a concept in general, it helps if you had some practical experience with it. Perhaps you had some difficulty getting started in formulating an initial solution to a problem, or perhaps you were unable to decide which of two algorithms was better. The goal of this book is to teach you how to make these decisions.

The study of algorithms includes many important and active areas of research. There are four distinct areas of study one can identify:

1. *How to devise algorithms* — Creating an algorithm is an art which may never be fully automated. A major goal of this book is to study vari-

ous design techniques that have proven to be useful in that they have often yielded good algorithms. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Many of the chapters of this book are organized around what we believe are the major methods of algorithm design. The reader may now wish to glance back at the table of contents to see what these methods are called. Some of these techniques may already be familiar, and some have been found to be so useful that books have been written about them. Dynamic programming is one such technique. Some of the techniques are especially useful in fields other than computer science such as operations research and electrical engineering. In this book we can only hope to give an introduction to these many approaches to algorithm formulation. All of the approaches we consider have applications in a variety of areas including computer science. But some important design techniques such as linear, nonlinear, and integer programming are not covered here as they are traditionally covered in other courses.

2. *How to validate algorithms* — Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as *algorithm validation*. The algorithm need not as yet be expressed as a program. It is sufficient to state it in any precise way. The purpose of the validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as *program proving* or sometimes as *program verification*. A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus. The second form is called a *specification*, and this may also be expressed in the predicate calculus. A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct. All these details may cause a proof to be very much longer than the program.

3. *How to analyze algorithms* — This field of study is called analysis of algorithms. As an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory (both immediate and auxiliary) to hold the program and data. *Analysis of algorithms* or *performance analysis* refers to the task of determining how much computing time and storage an algorithm requires. This is a challenging area which sometimes requires great mathematical skill. An important result of this study is that it allows you to make quantitative judgments about the value of one algorithm over another. Another result is that it allows you to predict whether the software will meet any efficiency constraints that exist.

Questions such as how well does an algorithm perform in the best case, in the worst case, or on the average are typical. For each algorithm in the text, an analysis is also given. Analysis is more fully described in Section 1.3.2.

4. *How to test a program* — Testing a program consists of two phases: debugging and profiling (or performance measurement). *Debugging* is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them. However, as E. Dijkstra has pointed out, "debugging can only point to the presence of errors, but not to their absence." In cases in which we cannot verify the correctness of output on sample data, the following strategy can be employed: let more than one programmer develop programs for the same problem, and compare the outputs produced by these programs. If the outputs match, then there is a good chance that they are correct. A proof of correctness is much more valuable than a thousand tests (if that proof is correct), since it guarantees that the program will work correctly for all possible inputs. *Profiling* or *performance measurement* is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results. These timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization. A description of the measurement of timing complexity can be found in Section 1.3.5. For some of the algorithms presented here, we show how to devise a range of data sets that will be useful for debugging and profiling.

These four categories serve to outline the questions we ask about algorithms throughout this book. As we can't hope to cover all these subjects completely, we content ourselves with concentrating on design and analysis, spending less time on program construction and correctness.

EXERCISES

1. Look up the words algorism and algorithm in your dictionary and write down their meanings.

2. The name al-Khowarizmi (algorithm) literally means "from the town of Khowarazm." This city is now known as Khiva, and is located in Uzbekistan. See if you can find this country in an atlas.

3. Use the WEB to find out more about al-Khowarizmi, e.g., his dates, a picture, or a stamp.

1.2 ALGORITHM SPECIFICATION

1.2.1 Pseudocode Conventions

In computational theory, we distinguish between an algorithm and a program. The latter does not have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a "wait" loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs always terminate, we use "algorithm" and "program" interchangeably in this text.

We can describe an algorithm in many ways. We can use a natural language like English, although if we select this option, we must make sure that the resulting instructions are definite. Graphic representations called *flowcharts* are another possibility, but they work well only if the algorithm is small and simple. In this text we present most of our algorithms using a pseudocode that resembles C and Pascal.

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces: { and }. A compound statement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by ;.

3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. We assume simple data types such as integer, float, char, boolean, and so on. Compound data types can be formed with **records**. Here is an example:

```
node = record
       {    datatype_1   data_1;
                   ⋮
            datatype_n   data_n;
            node         *link;
       }
```

   In this example, *link* is a pointer to the record type *node*. Individual data items of a record can be accessed with → and period. For instance if p points to a record of type *node*, $p \rightarrow data\_1$ stands for the value of the first field in the record. On the other hand, if q is a record of type *node*, $q.data\_1$ will denote its first field.

4. Assignment of values to variables is done using the assignment statement

   $\langle variable \rangle := \langle expression \rangle;$

5. There are two boolean values **true** and **false**. In order to produce these values, the logical operators **and**, **or**, and **not** and the relational operators $<, \leq, =, \neq, \geq$, and $>$ are provided.

6. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i, j)th element of the array is denoted as $A[i, j]$. Array indices start at zero.

7. The following looping statements are employed: **for**, **while**, and **repeat-until**. The **while** loop takes the following form:

   ```
   while ⟨condition⟩ do
   {
           ⟨statement 1⟩
                 ⋮
           ⟨statement n⟩
   }
   ```

   As long as $\langle condition \rangle$ is **true**, the statements get executed. When $\langle condition \rangle$ becomes **false**, the loop is exited. The value of $\langle condition \rangle$ is evaluated at the top of the loop.

   The general form of a **for** loop is

   ```
   for variable := value1 to value2 step step do
   {
           ⟨statement 1⟩
                 ⋮
           ⟨statement n⟩
   }
   ```

   Here *value*1, *value*2, and *step* are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause "**step** *step*" is optional and taken as +1 if it does not occur. *step* could either be positive or negative. *variable* is tested for termination at the start of each iteration. The **for** loop can be implemented as a **while** loop as follows:

```
$variable := value1$;
$fin := value2$;
$incr := step$;
while $((variable - fin) * step \leq 0)$ do
{
    $\langle statement\ 1 \rangle$
        ⋮
    $\langle statement\ n \rangle$
    $variable := variable + incr$;
}
```

A **repeat-until** statement is constructed as follows:

```
repeat
    $\langle statement\ 1 \rangle$
        ⋮
    $\langle statement\ n \rangle$
until $\langle condition \rangle$
```

The statements are executed as long as $\langle condition \rangle$ is **false**. The value of $\langle condition \rangle$ is computed after executing the statements.

The instruction **break;** can be used within any of the above looping instructions to force exit. In case of nested loops, **break;** results in the exit of the innermost loop that it is a part of. A **return** statement within any of the above also will result in exiting the loops. A **return** statement results in the exit of the function itself.

8. A conditional statement has the following forms:

```
if $\langle condition \rangle$ then $\langle statement \rangle$
if $\langle condition \rangle$ then $\langle statement\ 1 \rangle$ else $\langle statement\ 2 \rangle$
```

Here $\langle condition \rangle$ is a boolean expression and $\langle statement \rangle$, $\langle statement\ 1 \rangle$, and $\langle statement\ 2 \rangle$ are arbitrary statements (simple or compound).

We also employ the following **case** statement:

```
case
{
    :$\langle condition\ 1 \rangle$: $\langle statement\ 1 \rangle$
        ⋮
    :$\langle condition\ n \rangle$: $\langle statement\ n \rangle$
    :else: $\langle statement\ n+1 \rangle$
}
```

Here ⟨*statement* 1⟩, ⟨*statement* 2⟩, etc. could be either simple statements or compound statements. A **case** statement is interpreted as follows. If ⟨*condition* 1⟩ is **true**, ⟨*statement* 1⟩ gets executed and the **case** statement is exited. If ⟨*statement* 1⟩ is **false**, ⟨*condition* 2⟩ is evaluated. If ⟨*condition* 2⟩ is **true**, ⟨*statement* 2⟩ gets executed and the **case** statement exited, and so on. If none of the conditions ⟨*condition* 1⟩, ..., ⟨*condition n*⟩ are true, ⟨*statement n*+1⟩ is executed and the **case** statement is exited. The **else** clause is optional.

9. Input and output are done using the instructions **read** and **write**. No format is used to specify the size of input or output quantities.

10. There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and a body. The heading takes the form

    Algorithm *Name* (⟨*parameter list*⟩)

    where *Name* is the name of the procedure and (⟨*parameter list*⟩) is a listing of the procedure parameters. The body has one or more (simple or compound) statements enclosed within braces **{** and **}**. An algorithm may or may not return any values. Simple variables to procedures are passed by value. Arrays and records are passed by reference. An array name or a record name is treated as a pointer to the respective data type.

    As an example, the following algorithm finds and returns the maximum of n given numbers:

```
Algorithm Max($A$, $n$)
// $A$ is an array of size $n$.
{
    $Result := A[1]$;
    for $i := 2$ to $n$ do
        if $A[i] > Result$ then $Result := A[i]$;
    return $Result$;
}
```

    In this algorithm (named Max), A and n are procedure parameters. *Result* and i are local variables.

Next we present two examples to illustrate the process of translating a problem into an algorithm.

Example 1.1 [Selection sort] Suppose we must devise an algorithm that sorts a collection of $n \geq 1$ elements of arbitrary type. A simple solution is given by the following

From those elements that are currently unsorted, find the smallest and place it next in the sorted list.

Although this statement adequately describes the sorting problem, it is not an algorithm because it leaves several questions unanswered. For example, it does not tell us where and how the elements are initially stored or where we should place the result. We assume that the elements are stored in an array a, such that the ith integer is stored in the ith position $a[i]$, $1 \leq i \leq n$. Algorithm 1.1 is our first attempt at deriving a solution.

```
1   for $i := 1$ to $n$ do
2   {
3       Examine $a[i]$ to $a[n]$ and suppose
4       the smallest element is at $a[j]$;
5       Interchange $a[i]$ and $a[j]$;
6   }
```

Algorithm 1.1 Selection sort algorithm

To turn Algorithm 1.1 into a pseudocode program, two clearly defined subtasks remain: finding the smallest element (say $a[j]$) and interchanging it with $a[i]$. We can solve the latter problem using the code

$$t := a[i]; \; a[i] := a[j]; \; a[j] := t;$$

The first subtask can be solved by assuming the minimum is $a[i]$, checking $a[i]$ with $a[i+1], a[i+2], \ldots$, and, whenever a smaller element is found, regarding it as the new minimum. Eventually $a[n]$ is compared with the current minimum, and we are done. Putting all these observations together, we get the algorithm SelectionSort (Algorithm 1.2).

The obvious question to ask at this point is, Does SelectionSort work correctly? Throughout this text we use the notation $a[i : j]$ to denote the array elements $a[i]$ through $a[j]$.

Theorem 1.1 Algorithm SelectionSort(a, n) correctly sorts a set of $n \geq 1$ elements; the result remains in $a[1 : n]$ such that $a[1] \leq a[2] \leq \cdots \leq a[n]$.

Proof: We first note that for any i, say $i = q$, following the execution of lines 6 to 9, it is the case that $a[q] \leq a[r]$, $q < r \leq n$. Also observe that when i becomes greater than q, $a[1 : q]$ is unchanged. Hence, following the last execution of these lines (that is, $i = n$), we have $a[1] \leq a[2] \leq \cdots \leq a[n]$.

We observe at this point that the upper limit of the **for** loop in line 4 can be changed to $n - 1$ without damaging the correctness of the algorithm. □

```
Algorithm SelectionSort($a, n$)
// Sort the array $a[1 : n]$ into nondecreasing order.
{
    for $i := 1$ to $n$ do
    {
        $j := i$;
        for $k := i + 1$ to $n$ do
            if $(a[k] < a[j])$ then $j := k$;
        $t := a[i]$; $a[i] := a[j]$; $a[j] := t$;
    }
}
```

Algorithm 1.2 Selection sort

1.2.2 Recursive Algorithms

A recursive function is a function that is defined in terms of itself. Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is *direct recursive*. Algorithm $\mathcal{A}$ is said to be *indirect recursive* if it calls another algorithm which in turn calls $\mathcal{A}$. These recursive mechanisms are extremely powerful, but even more importantly, many times they can express an otherwise complex process very clearly. For these reasons we introduce recursion here.

Typically, beginning programmers view recursion as a somewhat mystical technique that is useful only for some very special class of problems (such as computing factorials or Ackermann's function). This is unfortunate because any algorithm that can be written using assignment, the **if-then-else** statement, and the **while** statement can also be written using assignment, the **if-then-else** statement, and recursion. Of course, this does not say that the resulting algorithm will necessarily be easier to understand. However, there are many instances when this will be the case. When is recursion an appropriate mechanism for algorithm exposition? One instance is when the problem itself is recursively defined. Factorial fits this category, as well as binomial coefficients, where

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} = \frac{n!}{m!(n-m)!}$$

The following two examples show how to develop a recursive algorithm. In the first example, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

Example 1.2 [Towers of Hanoi] The Towers of Hanoi puzzle is fashioned after the ancient Tower of Brahma ritual (see Figure 1.1). According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks. The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top. Besides this tower there were two other diamond towers (labeled B and C). Since the time of creation, Brahman priests have been attempting to move the disks from tower A to tower B using tower C for intermediate storage. As the disks are very heavy, they can be moved only one at a time. In addition, at no time can a disk be on top of a smaller disk. According to legend, the world will come to an end when the priests have completed their task.

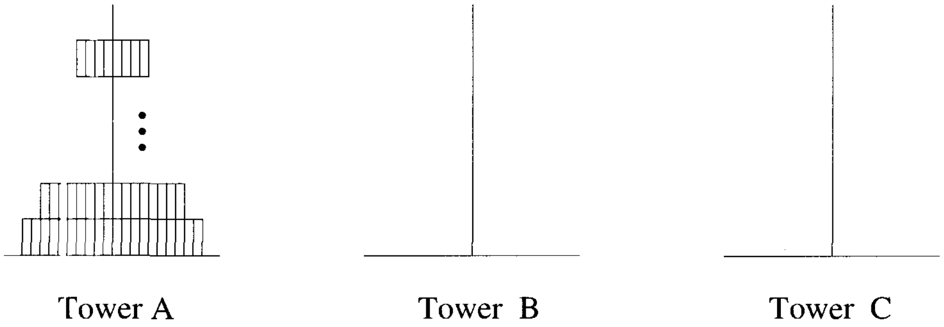

Figure 1.1 Towers of Hanoi

A very elegant solution results from the use of recursion. Assume that the number of disks is n. To get the largest disk to the bottom of tower B, we move the remaining $n-1$ disks to tower C and then move the largest to tower B. Now we are left with the task of moving the disks from tower C to tower B. To do this, we have towers A and B available. The fact that tower B has a disk on it can be ignored as the disk is larger than the disks being moved from tower C and so any disk can be placed on top of it. The recursive nature of the solution is apparent from Algorithm 1.3. This algorithm is invoked by TowersOfHanoi(n,A,B,C). Observe that our solution for an n-disk problem is formulated in terms of solutions to two $(n-1)$-disk problems. □

Example 1.3 [Permutation generator] Given a set of $n \geq 1$ elements, the problem is to print all possible permutations of this set. For example, if the set is $\{a, b, c\}$, then the set of permutations is $\{(a, b, c), (a, c, b), (b, a, c),$

```
Algorithm TowersOfHanoi($n, x, y, z$)
// Move the top $n$ disks from tower $x$ to tower $y$.
{
    if ($n \geq 1$) then
    {
        TowersOfHanoi($n-1, x, z, y$);
        write ("move top disk from tower", $x$,
        "to top of tower", $y$);
        TowersOfHanoi($n-1, z, y, x$);
    }
}
```

Algorithm 1.3 Towers of Hanoi

(b, c, a), (c, a, b), $(c, b, a)\}$. It is easy to see that given n elements, there are $n!$ different permutations. A simple algorithm can be obtained by looking at the case of four elements (a, b, c, d). The answer can be constructed by writing

1. a followed by all the permutations of (b, c, d)
2. b followed by all the permutations of (a, c, d)
3. c followed by all the permutations of (a, b, d)
4. d followed by all the permutations of (a, b, c)

The expression "followed by all the permutations" is the clue to recursion. It implies that we can solve the problem for a set with n elements if we have an algorithm that works on $n-1$ elements. These considerations lead to Algorithm 1.4, which is invoked by Perm$(a, 1, n)$. Try this algorithm out on sets of length one, two, and three to ensure that you understand how it works. □

EXERCISES

1. Horner's rule is a means for evaluating a polynomial at a point x_0 using a minimum number of multiplications. If the polynomial is $A(x)$ $= a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, Horner's rule is

```
Algorithm Perm($a, k, n$)
{
    if ($k = n$) then write ($a[1 : n]$); // Output permutation.
    else // $a[k : n]$ has more than one permutation.
        // Generate these recursively.
        for $i := k$ to $n$ do
        {
            $t := a[k]$; $a[k] := a[i]$; $a[i] := t$;
            Perm($a, k + 1, n$);
            // All permutations of $a[k + 1 : n]$
            $t := a[k]$; $a[k] := a[i]$; $a[i] := t$;
        }
}
```

Algorithm 1.4 Recursive permutation generator

$$A(x_0) \;=\; (\cdots\,(a_n x_0 + a_{n-1})x_0 + \cdots + a_1)x_0 + a_0$$

Write an algorithm to evaluate a polynomial using Horner's rule.

2. Given n boolean variables $x_1, x_2, \ldots,$ and x_n, we wish to print all possible combinations of truth values they can assume. For instance, if $n = 2$, there are four possibilities: true, true; true, false; false, true; and false, false. Write an algorithm to accomplish this.

3. Devise an algorithm that inputs three integers and outputs them in nondecreasing order.

4. Present an algorithm that searches an unsorted array $a[1 : n]$ for the element x. If x occurs, then return a position in the array; else return zero.

5. The factorial function $n!$ has value 1 when $n \leq 1$ and value $n * (n-1)!$ when $n > 1$. Write both a recursive and an iterative algorithm to compute $n!$.

6. The Fibonacci numbers are defined as $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$. Write both a recursive and an iterative algorithm to compute f_i.

7. Give both a recursive and an iterative algorithm to compute the binomial coefficient $\binom{n}{m}$ as defined in Section 1.2.2, where $\binom{n}{0} = \binom{n}{n} = 1$.

8. Ackermann's function $A(m, n)$ is defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,\ 1) & \text{if } n = 0 \\ A(m-1,\ A(m,\ n-1)) & \text{otherwise} \end{cases}$$

   This function is studied because it grows very fast for small values of m and n. Write a recursive algorithm for computing this function. Then write a nonrecursive algorithm for computing it.

9. The *pigeonhole principle* states that if a function f has n distinct inputs but less than n distinct outputs, then there exist two inputs a and b such that $a \neq b$ and $f(a) = f(b)$. Present an algorithm to find a and b such that $f(a) = f(b)$. Assume that the function inputs are $1, 2, \ldots,$ and n.

10. Give an algorithm to solve the following problem: Given n, a positive integer, determine whether n is the sum of all of its divisors, that is, whether n is the sum of all t such that $1 \leq t < n$, and t divides n.

11. Consider the function $F(x)$ that is defined by "if x is even, then $F(x) = x/2$; else $F(x) = F(F(3x + 1))$." Prove that $F(x)$ terminates for all integers x. (*Hint:* Consider integers of the form $(2i + 1)2^k - 1$ and use induction.)

12. If S is a set of n elements, the *powerset* of S is the set of all possible subsets of S. For example, if $S = (a, b, c)$, then $powerset(S) = \{(\ ), (a), (b), (c), (a, b), (a, c), (b, c), (a, b, c)\}$. Write a recursive algorithm to compute $powerset(S)$.

1.3 PERFORMANCE ANALYSIS

One goal of this book is to develop skills for making evaluative judgments about algorithms. There are many criteria upon which we can judge an algorithm. For instance:

1. Does it do what we want it to do?

2. Does it work correctly according to the original specifications of the task?

3. Is there documentation that describes how to use it and how it works?

4. Are procedures created in such a way that they perform logical subfunctions?

5. Is the code readable?

These criteria are all vitally important when it comes to writing software, most especially for large systems. Though we do not discuss how to reach these goals, we try to achieve them throughout this book with the pseudocode algorithms we write. Hopefully this more subtle approach will gradually infect your own program-writing habits so that you will automatically strive to achieve these goals.

There are other criteria for judging algorithms that have a more direct relationship to performance. These have to do with their computing time and storage requirements.

Definition 1.2 [Space/Time complexity] The *space complexity* of an algorithm is the amount of memory it needs to run to completion. The *time complexity* of an algorithm is the amount of computer time it needs to run to completion. □

Performance evaluation can be loosely divided into two major phases: (1) a priori estimates and (2) a posteriori testing. We refer to these as *performance analysis* and *performance measurement* respectively.

1.3.1 Space Complexity

Algorithm abc (Algorithm 1.5) computes $a+b+b*c+(a+b-c)/(a+b)+4.0$; Algorithm Sum (Algorithm 1.6) computes $\sum_{i=1}^{n} a[i]$ iteratively, where the $a[i]$'s are real numbers; and RSum (Algorithm 1.7) is a recursive algorithm that computes $\sum_{i=1}^{n} a[i]$.

```
Algorithm abc($a, b, c$)
{
    return $a + b + b * c + (a + b - c)/(a + b) + 4.0$;
}
```

Algorithm 1.5 Computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$

The space needed by each of these algorithms is seen to be the sum of the following components:

```
Algorithm Sum($a, n$)
{
    $s := 0.0$;
    for $i := 1$ to $n$ do
        $s := s + a[i]$;
    return $s$;
}
```

Algorithm 1.6 Iterative function for sum

```
Algorithm RSum($a, n$)
{
    if ($n \leq 0$) then return $0.0$;
    else return RSum($a, n - 1$) + $a[n]$;
}
```

Algorithm 1.7 Recursive function for sum

1. A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called *aggregate*), space for constants, and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (insofar as this space depends on the instance characteristics).

The space requirement $S(P)$ of any algorithm P may therefore be written as $S(P) = c + S_P$(instance characteristics), where c is a constant.

When analyzing the space complexity of an algorithm, we concentrate solely on estimating S_P(instance characteristics). For any given problem, we need first to determine which instance characteristics to use to measure the space requirements. This is very problem specific, and we resort to examples to illustrate the various possibilities. Generally speaking, our choices are limited to quantities related to the number and magnitude of the inputs to and outputs from the algorithm. At times, more complex measures of the interrelationships among the data items are used.

Example 1.4 For Algorithm 1.5, the problem instance is characterized by the specific values of a, b, and c. Making the assumption that one word is adequate to store the values of each of a, b, c, and the result, we see that the space needed by abc is independent of the instance characteristics. Consequently, S_P(instance characteristics) $= 0$. □

Example 1.5 The problem instances for Algorithm 1.6 are characterized by n, the number of elements to be summed. The space needed by n is one word, since it is of type *integer*. The space needed by a is the space needed by variables of type array of floating point numbers. This is at least n words, since a must be large enough to hold the n elements to be summed. So, we obtain $S_{\mathsf{Sum}}(n) \geq (n+3)$ (n for $a[\ ]$, one each for n, i, and s). □

Example 1.6 Let us consider the algorithm RSum (Algorithm 1.7). As in the case of Sum, the instances are characterized by n. The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to RSum requires at least three words (including space for the values of n, the return address, and a pointer to $a[\ ]$). Since the depth of recursion is $n+1$, the recursion stack space needed is $\geq 3(n+1)$. □

1.3.2 Time Complexity

The time $T(P)$ taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we concern ourselves with just the run time of a program. This run time is denoted by t_P(instance characteristics).

Because many of the factors t_P depends on are not known at the time a program is conceived, it is reasonable to attempt only to estimate t_P. If we knew the characteristics of the compiler to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code for P. So, we could obtain an expression for $t_P(n)$ of the form

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

where n denotes the instance characteristics, and c_a, c_s, c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on, and ADD, SUB, MUL, DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

Obtaining such an exact formula is in itself an impossible task, since the time needed for an addition, subtraction, multiplication, and so on, often depends on the numbers being added, subtracted, multiplied, and so on. The value of $t_P(n)$ for any given n can be obtained only experimentally. The program is typed, compiled, and run on a particular machine. The execution time is physically clocked, and $t_P(n)$ obtained. Even with this experimental approach, one could face difficulties. In a multiuser system, the execution time depends on such factors as system load, the number of other programs running on the computer at the time program P is run, the characteristics of these other programs, and so on.

Given the minimal utility of determining the exact number of additions, subtractions, and so on, that are needed to solve a problem instance with characteristics given by n, we might as well lump all the operations together (provided that the time required by each is relatively independent of the instance characteristics) and obtain a count for the total number of operations. We can go one step further and count only the number of program steps.

A *program step* is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement

$$\textbf{return}\ a + b + b * c + (a + b - c)/(a + b) + 4.0;$$

of Algorithm 1.5 could be regarded as a step since its execution time is independent of the instance characteristics (this statement is not strictly true, since the time for a multiply and divide generally depends on the numbers involved in the operation).

The number of steps any program statement is assigned depends on the kind of statement. For example, comments count as zero steps; an assignment statement which does not involve any calls to other algorithms is counted as one step; in an iterative statement such as the **for**, **while**, and **repeat-until** statements, we consider the step counts only for the control part of the statement. The control parts for **for** and **while** statements have the following forms:

```
for $i := \langle expr \rangle$ to $\langle expr1 \rangle$ do

while $(\langle expr \rangle)$ do
```

Each execution of the control part of a **while** statement is given a step count equal to the number of step counts assignable to $\langle expr \rangle$. The step count for each execution of the control part of a **for** statement is one, unless the counts attributable to $\langle expr \rangle$ and $\langle expr1 \rangle$ are functions of the instance characteristics. In this latter case, the first execution of the control part of the **for** has a step count equal to the sum of the counts for $\langle expr \rangle$ and $\langle expr1 \rangle$ (note that these expressions are computed only when the loop is started). Remaining executions of the **for** statement have a step count of one; and so on.

We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways. In the first method, we introduce a new variable, *count*, into the program. This is a global variable with initial value 0. Statements to increment *count* by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executed, *count* is incremented by the step count of that statement.

Example 1.7 When the statements to increment *count* are introduced into Algorithm 1.6, the result is Algorithm 1.8. The change in the value of *count* by the time this program terminates is the number of steps executed by Algorithm 1.6.

Since we are interested in determining only the change in the value of *count*, Algorithm 1.8 may be simplified to Algorithm 1.9. For every initial value of *count*, Algorithms 1.8 and 1.9 compute the same final value for *count*. It is easy to see that in the **for** loop, the value of *count* will increase by a total of $2n$. If *count* is zero to start with, then it will be $2n + 3$ on termination. So each invocation of Sum (Algorithm 1.6) executes a total of $2n + 3$ steps. □

```
Algorithm Sum($a, n$)
{
    $s := 0.0$;
    $count := count + 1$; // $count$ is global; it is initially zero.
    for $i := 1$ to $n$ do
    {
        $count := count + 1$; // For for
        $s := s + a[i]$; $count := count + 1$; // For assignment
    }
    $count := count + 1$; // For last time of for
    $count := count + 1$; // For the return
    return $s$;
}
```

Algorithm 1.8 Algorithm 1.6 with count statements added

```
Algorithm Sum($a, n$)
{
    for $i := 1$ to $n$ do $count := count + 2$;
    $count := count + 3$;
}
```

Algorithm 1.9 Simplified version of Algorithm 1.8

Example 1.8 When the statements to increment *count* are introduced into Algorithm 1.7, Algorithm 1.10 is obtained. Let $t_{\mathsf{RSum}}(n)$ be the increase in the value of *count* when Algorithm 1.10 terminates. We see that $t_{\mathsf{RSum}}(0) = 2$. When $n > 0$, *count* increases by 2 plus whatever increase results from the invocation of RSum from within the **else** clause. From the definition of t_{RSum}, it follows that this additional increase is $t_{\mathsf{RSum}}(n-1)$. So, if the value of *count* is zero initially, its value at the time of termination is $2+t_{\mathsf{RSum}}(n-1)$, $n > 0$.

```
Algorithm RSum($a, n$)
{
    $count := count + 1$; // For the if conditional
    if ($n \leq 0$) then
    {
        $count := count + 1$; // For the return
        return 0.0;
    }
    else
    {
        $count := count + 1$;  // For the addition, function
                               // invocation and return
        return RSum($a, n-1$) + $a[n]$;
    }
}
```

Algorithm 1.10 Algorithm 1.7 with count statements added

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example,

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

These recursive formulas are referred to as *recurrence relations*. One way of solving any such recurrence relation is to make repeated substitutions for each occurrence of the function t_{RSum} on the right-hand side until all such occurrences disappear:

$$\begin{aligned} t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\ &= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\ &= 2(2) + t_{\mathsf{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\mathsf{RSum}}(0) \\ &= 2n + 2, \qquad\qquad n \geq 0 \end{aligned}$$

So the step count for RSum (Algorithm 1.7) is $2n+2$. □

The step count is useful in that it tells us how the run time for a program changes with changes in the instance characteristics. From the step count for Sum, we see that if n is doubled, the run time also doubles (approximately); if n increases by a factor of 10, the run time increases by a factor of 10; and so on. So, the run time grows *linearly* in n. We say that Sum is a linear time algorithm (the time complexity is linear in the instance characteristic n).

Definition 1.3 [Input size] One of the instance characteristics that is frequently used in the literature is the *input size*. The input size of any instance of a problem is defined to be the number of words (or the number of elements) needed to describe that instance. The input size for the problem of summing an array with n elements is $n+1$, n for listing the n elements and 1 for the value of n (Algorithms 1.6 and 1.7). The problem tackled in Algorithm 1.5 has an input size of 3. If the input to any problem instance is a single element, the input size is normally taken to be the number of bits needed to specify that element. Run times for many of the algorithms presented in this text are expressed as functions of the corresponding input sizes. □

Example 1.9 [Matrix addition] Algorithm 1.11 is to add two $m \times n$ matrices a and b together. Introducing the *count*-incrementing statements leads to Algorithm 1.12. Algorithm 1.13 is a simplified version of Algorithm 1.12 that computes the same value for *count*. Examining Algorithm 1.13, we see that line 7 is executed n times for each value of i, or a total of mn times; line 5 is executed m times; and line 9 is executed once. If *count* is 0 to begin with, it will be $2mn + 2m + 1$ when Algorithm 1.13 terminates.

From this analysis we see that if $m > n$, then it is better to interchange the two **for** statements in Algorithm 1.11. If this is done, the step count becomes $2mn+2n+1$. Note that in this example the instance characteristics are given by m and n and the input size is $2mn+2$. □

The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. This figure is often arrived at by first determining the number of

```
Algorithm Add($a, b, c, m, n$)
{
    for $i := 1$ to $m$ do
        for $j := 1$ to $n$ do
            $c[i, j] := a[i, j] + b[i, j]$;
}
```

Algorithm 1.11 Matrix addition

```
Algorithm Add($a, b, c, m, n$)
{
    for $i := 1$ to $m$ do
    {
        $count := count + 1$; // For 'for $i$'
        for $j := 1$ to $n$ do
        {
            $count := count + 1$; // For 'for $j$'
            $c[i, j] := a[i, j] + b[i, j]$;
            $count := count + 1$; // For the assignment
        }
        $count := count + 1$;// For loop initialization and
                             // last time of 'for $j$'
    }
    $count := count + 1$;      // For loop initialization and
                             // last time of 'for $i$'
}
```

Algorithm 1.12 Matrix addition with counting statements

```
Algorithm Add($a, b, c, m, n$)
{
    for $i := 1$ to $m$ do
    {
        $count := count + 2$;
        for $j := 1$ to $n$ do
            $count := count + 2$;
    }
    $count := count + 1$;
}
```

Algorithm 1.13 Simplified algorithm with counting only

steps per execution (s/e) of the statement and the total number of times (i.e., frequency) each statement is executed. *The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.* By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

In Table 1.1, the number of steps per execution and the frequency of each of the statements in Sum (Algorithm 1.6) have been listed. The total number of steps required by the algorithm is determined to be $2n + 3$. It is important to note that the frequency of the **for** statement is $n + 1$ and not n. This is so because i has to be incremented to $n + 1$ before the **for** loop can terminate.

Table 1.2 gives the step count for RSum (Algorithm 1.7). Notice that under the s/e (steps per execution) column, the **else** clause has been given a count of $1 + t_{\mathsf{RSum}}(n - 1)$. This is the total cost of this line each time it is executed. It includes all the steps that get executed as a result of the invocation of RSum from the **else** clause. The frequency and total steps columns have been split into two parts: one for the case $n = 0$ and the other for the case $n > 0$. This is necessary because the frequency (and hence total steps) for some statements is different for each of these cases.

Table 1.3 corresponds to algorithm Add (Algorithm 1.11). Once again, note that the frequency of the first **for** loop is $m + 1$ and not m. This is so as i needs to be incremented up to $m + 1$ before the loop can terminate. Similarly, the frequency for the second **for** loop is $m(n + 1)$.

When you have obtained sufficient experience in computing step counts, you can avoid constructing the frequency table and obtain the step count as in the following example.

Statement	s/e	frequency	total steps
1 **Algorithm** $\mathsf{Sum}(a, n)$	0	$-$	0
2 {	0	$-$	0
3 $s := 0.0;$	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$n+1$
5 $s := s + a[i];$	1	n	n
6 **return** s;	1	1	1
7 }	0	$-$	0
Total			$2n+3$

Table 1.1 Step table for Algorithm 1.6

		frequency		total steps	
Statement	s/e	$n=0$	$n>0$	$n=0$	$n>0$
1 **Algorithm** $\mathsf{RSum}(a, n)$	0	$-$	$-$	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 $\mathsf{RSum}(a, n-1) + a[n];$	$1+x$	0	1	0	$1+x$
7 }	0	$-$	$-$	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

Table 1.2 Step table for Algorithm 1.7

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m + 1$	$m + 1$
4 **for** $j := 1$ **to** n **do**	1	$m(n + 1)$	$mn + m$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	mn	mn
6 }	0	–	0
Total			$2mn + 2m + 1$

Table 1.3 Step table for Algorithm 1.11

Example 1.10 [Fibonacci numbers] The Fibonacci sequence of numbers starts as

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Each new term is obtained by taking the sum of the two previous terms. If we call the first term of the sequence f_0, then $f_0 = 0$, $f_1 = 1$, and in general

$$f_n = f_{n-1} + f_{n-2}, \quad n \geq 2$$

Fibonacci (Algorithm 1.14) takes as input any nonnegative integer n and prints the value f_n.

To analyze the time complexity of this algorithm, we need to consider the two cases (1) $n = 0$ or 1 and (2) $n > 1$. When $n = 0$ or 1, lines 4 and 5 get executed once each. Since each line has an s/e of 1, the total step count for this case is 2. When $n > 1$, lines 4, 8, and 14 are each executed once. Line 9 gets executed n times, and lines 11 and 12 get executed $n - 1$ times each (note that the last time line 9 is executed, i is incremented to $n + 1$, and the loop exited). Line 8 has an s/e of 2, line 12 has an s/e of 2, and line 13 has an s/e of 0. The remaining lines that get executed have s/e's of 1. The total steps for the case $n > 1$ is therefore $4n + 1$. □

Summary of Time Complexity

The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for. The number of steps is itself a function of the instance characteristics. Although any specific instance may have several characteristics (e.g., the number of inputs, the number of outputs, the magnitudes of the inputs and outputs), the number

```
Algorithm Fibonacci($n$)
// Compute the $n$th Fibonacci number.
{
    if ($n \leq 1$) then
        write ($n$);
    else
    {
        $fnm2 := 0$; $fnm1 := 1$;
        for $i := 2$ to $n$ do
        {
            $fn := fnm1 + fnm2$;
            $fnm2 := fnm1$; $fnm1 := fn$;
        }
        write ($fn$);
    }
}
```

Algorithm 1.14 Fibonacci numbers

of steps is computed as a function of some subset of these. Usually, we choose those characteristics that are of importance to us. For example, we might wish to know how the computing (or run) time (i.e., time complexity) increases as the number of inputs increase. In this case the number of steps will be computed as a function of the number of inputs alone. For a different algorithm, we might be interested in determining how the computing time increases as the magnitude of one of the inputs increases. In this case the number of steps will be computed as a function of the magnitude of this input alone. Thus, before the step count of an algorithm can be determined, we need to know exactly which characteristics of the problem instance are to be used. These define the variables in the expression for the step count. In the case of Sum, we chose to measure the time complexity as a function of the number n of elements being added. For algorithm Add, the choice of characteristics was the number m of rows and the number n of columns in the matrices being added.

Once the relevant characteristics $(n, m, p, q, r, \ldots)$ have been selected, we can define what a step is. A *step* is any computation unit that is independent of the characteristics $(n, m, p, q, r, \ldots)$. Thus, 10 additions can be one step; 100 multiplications can also be one step; but n additions cannot. Nor can $m/2$ additions, $p + q$ subtractions, and so on, be counted as one step.

A systematic way to assign step counts was also discussed. Once this has been done, the time complexity (i.e., the total step count) of an algorithm can be obtained using either of the two methods discussed.

The examples we have looked at so far were sufficiently simple that the time complexities were nice functions of fairly simple characteristics like the number of inputs and the number of rows and columns. For many algorithms, the time complexity is not dependent solely on the number of inputs or outputs or some other easily specified characteristic. For example, the searching algorithm you wrote for Exercise 4 in Section 1.2, may terminate in one step if x is the first element examined by your algorithm, or it may take two steps (this happens if x is the second element examined), and so on. In other words, knowing n alone is not enough to estimate the run time of your algorithm.

We can extricate ourselves from the difficulties resulting from situations when the chosen parameters are not adequate to determine the step count uniquely by defining three kinds of step counts: best case, worst case, and average. The *best-case step count* is the minimum number of steps that can be executed for the given parameters. The *worst-case step count* is the maximum number of steps that can be executed for the given parameters. The *average step count* is the average number of steps executed on instances with the given parameters.

Our motivation to determine step counts is to be able to compare the time complexities of two algorithms that compute the same function and also to predict the growth in run time as the instance characteristics change.

Determining the exact step count (best case, worst case, or average) of an algorithm can prove to be an exceedingly difficult task. Expending immense effort to determine the step count exactly is not a very worthwhile endeavor, since the notion of a step is itself inexact. (Both the instructions $x := y;$ and $x := y + z + (x/y) + (x * y * z - x/z);$ count as one step.) Because of the inexactness of what a step stands for, the exact step count is not very useful for comparative purposes. An exception to this is when the difference between the step counts of two algorithms is very large, as in $3n + 3$ versus $100n + 10$. We might feel quite safe in predicting that the algorithm with step count $3n+3$ will run in less time than the one with step count $100n+10$. But even in this case, it is not necessary to know that the exact step count is $100n + 10$. Something like, "it's about $80n$ or $85n$ or $75n$," is adequate to arrive at the same conclusion.

For most situations, it is adequate to be able to make a statement like $c_1n^2 \leq t_P(n) \leq c_2n^2$ or $t_Q(n,m) = c_1n + c_2m$, where c_1 and c_2 are nonnegative constants. This is so because if we have two algorithms with a complexity of $c_1n^2 + c_2n$ and c_3n respectively, then we know that the one with complexity c_3n will be faster than the one with complexity $c_1n^2 + c_2n$ for sufficiently large values of n. For small values of n, either algorithm could be faster (depending on c_1, c_2, and c_3). If $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$, then

$c_1n^2 + c_2n \le c_3n$ for $n \le 98$ and $c_1n^2 + c_2n > c_3n$ for $n > 98$. If $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$, then $c_1n^2 + c_2n \le c_3n$ for $n \le 998$.

No matter what the values of c_1, c_2, and c_3, there will be an n beyond which the algorithm with complexity c_3n will be faster than the one with complexity $c_1n^2 + c_2n$. This value of n will be called the *break-even point.* If the break-even point is zero, then the algorithm with complexity c_3n is always faster (or at least as fast). The exact break-even point cannot be determined analytically. The algorithms have to be run on a computer in order to determine the break-even point. To know that there is a break-even point, it is sufficient to know that one algorithm has complexity $c_1n^2 + c_2n$ and the other c_3n for some constants c_1, c_2, and c_3. There is little advantage in determining the exact values of c_1, c_2, and c_3.

1.3.3 Asymptotic Notation (O, Ω, Θ)

With the previous discussion as motivation, we introduce some terminology that enables us to make meaningful (but inexact) statements about the time and space complexities of an algorithm. In the remainder of this chapter, the functions f and g are nonnegative functions.

Definition 1.4 [Big “oh"] The function $f(n) = O(g(n))$ (read as “f of n is big oh of g of n”) iff (if and only if) there exist positive constants c and n_0 such that $f(n) \le c * g(n)$ for all n, $n \ge n_0$. □

Example 1.11 The function $3n + 2 = O(n)$ as $3n + 2 \le 4n$ for all $n \ge 2$. $3n + 3 = O(n)$ as $3n + 3 \le 4n$ for all $n \ge 3$. $100n + 6 = O(n)$ as $100n + 6 \le 101n$ for all $n \ge 6$. $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \le 11n^2$ for all $n \ge 5$. $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \le 1001n^2$ for $n \ge 100$. $6 * 2^n + n^2 = O(2^n)$ as $6 * 2^n + n^2 \le 7 * 2^n$ for $n \ge 4$. $3n + 3 = O(n^2)$ as $3n + 3 \le 3n^2$ for $n \ge 2$. $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \le 10n^4$ for $n \ge 2$. $3n + 2 \ne O(1)$ as $3n + 2$ is not less than or equal to c for any constant c and all $n \ge n_0$. $10n^2 + 4n + 2 \ne O(n)$. □

We write $O(1)$ to mean a computing time that is a constant. $O(n)$ is called *linear*, $O(n^2)$ is called *quadratic*, $O(n^3)$ is called *cubic*, and $O(2^n)$ is called *exponential.* If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n, than if it had taken $O(n)$. Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$. These seven computing times–$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$–are the ones we see most often in this book.

As illustrated by the previous example, the statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \ge n_0$. It does not say anything about how good this bound is. Notice that $n = O(2^n)$, $n = O(n^{2.5})$, $n = O(n^3)$, $n = O(2^n)$, and so on. For the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$. So, while we often say that $3n + 3 = O(n)$, we almost never say that $3n + 3 = O(n^2)$, even though this latter statement is correct.

From the definition of O, it should be clear that $f(n) = O(g(n))$ is not the same as $O(g(n)) = f(n)$. In fact, it is meaningless to say that $O(g(n)) = f(n)$. The use of the symbol $=$ is unfortunate because this symbol commonly denotes the equals relation. Some of the confusion that results from the use of this symbol (which is standard terminology) can be avoided by reading the symbol $=$ as "is" and not as "equals."

Theorem 1.2 obtains a very useful result concerning the order of $f(n)$ (that is, the $g(n)$ in $f(n) = O(g(n))$) when $f(n)$ is a polynomial in n.

Theorem 1.2 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof:

$$\begin{array}{rcll} f(n) & \leq & \sum_{i=0}^{m} |a_i| n^i & \\ & \leq & n^m \sum_{i=0}^{m} |a_i| n^{i-m} & \\ & \leq & n^m \sum_{i=0}^{m} |a_i| & \text{for } n \geq 1 \end{array}$$

So, $f(n) = O(n^m)$ (assuming that m is fixed). □

Definition 1.5 [Omega] The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$. □

Example 1.12 The function $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (the inequality holds for $n \geq 0$, but the definition of Ω requires an $n_0 > 0$). $3n+3 = \Omega(n)$ as $3n+3 \geq 3n$ for $n \geq 1$. $100n+6 = \Omega(n)$ as $100n+6 \geq 100n$ for $n \geq 1$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$. $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$. Observe also that $3n + 3 = \Omega(1)$, $10n^2 + 4n + 2 = \Omega(n)$, $10n^2 + 4n + 2 = \Omega(1)$, $6 * 2^n + n^2 = \Omega(n^{100})$, $6 * 2^n + n^2 = \Omega(n^{50.2})$, $6 * 2^n + n^2 = \Omega(n^2)$, $6 * 2^n + n^2 = \Omega(n)$, and $6 * 2^n + n^2 = \Omega(1)$. □

As in the case of the big oh notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. The function $g(n)$ is only a lower bound on $f(n)$. For the statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true. So, while we say that $3n + 3 = \Omega(n)$ and $6 * 2^n + n^2 = \Omega(2^n)$, we almost never say that $3n + 3 = \Omega(1)$ or $6 * 2^n + n^2 = \Omega(1)$, even though both of these statements are correct.

Theorem 1.3 is the analogue of Theorem 1.2 for the omega notation.

Theorem 1.3 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Proof: Left as an exercise. □

Definition 1.6 [Theta] The function $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$. □

Example 1.13 The function $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$. $3n + 3 = \Theta(n)$, $10n^2 + 4n + 2 = \Theta(n^2)$, $6 * 2^n + n^2 = \Theta(2^n)$, and $10 * \log n + 4 = \Theta(\log n)$. $3n + 2 \neq \Theta(1)$, $3n + 3 \neq \Theta(n^2)$, $10n^2 + 4n + 2 \neq \Theta(n)$, $10n^2 + 4n + 2 \neq \Theta(1)$, $6 * 2^n + n^2 \neq \Theta(n^2)$, $6 * 2^n + n^2 \neq \Theta(n^{100})$, and $6 * 2^n + n^2 \neq \Theta(1)$. □

The theta notation is more precise than both the the big oh and omega notations. The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

Notice that the coefficients in all of the $g(n)$'s used in the preceding three examples have been 1. This is in accordance with practice. We almost never find ourselves saying that $3n + 3 = O(3n)$, that $10 = O(100)$, that $10n^2 + 4n + 2 = \Omega(4n^2)$, that $6 * 2^n + n^2 = O(6 * 2^n)$, or that $6 * 2^n + n^2 = \Theta(4 * 2^n)$, even though each of these statements is true.

Theorem 1.4 If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Proof: Left as an exercise. □

Definition 1.7 [Little "oh"] The function $f(n) = o(g(n))$ (read as "f of n is little oh of g of n") iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

□

Example 1.14 The function $3n + 2 = o(n^2)$ since $\lim_{n \to \infty} \frac{3n+2}{n^2} = 0$. $3n + 2 = o(n \log n)$. $3n + 2 = o(n \log \log n)$. $6 * 2^n + n^2 = o(3^n)$. $6 * 2^n + n^2 = o(2^n \log n)$. $3n + 2 \neq o(n)$. $6 * 2^n + n^2 \neq o(2^n)$. □

Analogous to o is the notation ω defined as follows.

Definition 1.8 [Little omega] The function $f(n) = \omega(g(n))$ (read as "f of n is little omega of g of n") iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

□

Example 1.15 Let us reexamine the time complexity analyses of the previous section. For the algorithm Sum (Algorithm 1.6) we determined that $t_{\mathsf{Sum}}(n) = 2n + 3$. So, $t_{\mathsf{Sum}}(n) = \Theta(n)$. For Algorithm 1.7, $t_{\mathsf{RSum}}(n) = 2n + 2 = \Theta(n)$. □

Although we might all see that the O, Ω, and Θ notations have been used correctly in the preceding paragraphs, we are still left with the question, Of what use are these notations if we have to first determine the step count exactly? The answer to this question is that the asymptotic complexity (i.e., the complexity in terms of O, Ω, and Θ) can be determined quite easily without determining the exact step count. This is usually done by first determining the asymptotic complexity of each statement (or group of statements) in the algorithm and then adding these complexities. Tables 1.4 through 1.6 do just this for Sum, RSum, and Add (Algorithms 1.6, 1.7, and 1.11).

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 $s := 0.0;$	1	1	$\Theta(1)$
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$\Theta(n)$
5 $s := s + a[i];$	1	n	$\Theta(n)$
6 **return** $s;$	1	1	$\Theta(1)$
7 }	0	–	$\Theta(0)$
Total			$\Theta(n)$

Table 1.4 Asymptotic complexity of Sum (Algorithm 1.6)

Although the analyses of Tables 1.4 through 1.6 are carried out in terms of step counts, it is correct to interpret $t_P(n) = \Theta(g(n))$, $t_P(n) = \Omega(g(n))$, or $t_P(n) = O(g(n))$ as a statement about the computing time of algorithm P. This is so because each step takes only $\Theta(1)$ time to execute.

Statement	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 **Algorithm** RSum(a, n)	0	–	–	0	$\Theta(0)$
2 {	0	–	–	0	$\Theta(0)$
3 **if** $(n \leq 0)$ **then**	1	1	1	1	$\Theta(1)$
4 **return** 0.0;	1	1	0	1	$\Theta(0)$
5 **else return**					
6 RSum$(a, n-1) + a[n]$);	$1+x$	0	1	0	$\Theta(1+x)$
7 }	0	–	–	0	$\Theta(0)$
Total				2	$\Theta(1+x)$

$$x = t_{\mathsf{RSum}}(n-1)$$

Table 1.5 Asymptotic complexity of RSum (Algorithm 1.7).

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	$\Theta(0)$
2 {	0	–	$\Theta(0)$
3 **for** $i := 1$ **to** m **do**	1	$\Theta(m)$	$\Theta(m)$
4 **for** $i := 1$ **to** n **do**	1	$\Theta(mn)$	$\Theta(mn)$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	$\Theta(mn)$	$\Theta(mn)$
6 }	0	–	$\Theta(0)$
Total			$\Theta(mn)$

Table 1.6 Asymptotic complexity of Add (Algorithm 1.11)

After you have had some experience using the table method, you will be in a position to arrive at the asymptotic complexity of an algorithm by taking a more global approach. We elaborate on this method in the following examples.

Example 1.16 [Permutation generator] Consider Perm (Algorithm 1.4). When $k = n$, we see that the time taken is $\Theta(n)$. When $k < n$, the **else** clause is entered. At this time, the second **for** loop is entered $n - k + 1$ times. Each iteration of this loop takes $\Theta(n + t_{\mathsf{Perm}}(k+1, n))$ time. So, $t_{\mathsf{Perm}}(k,\ n) = \Theta((n-k+1)(n+t_{\mathsf{Perm}}(k+1,\ n)))$ when $k\ <\ n$. Since $t_{\mathsf{Perm}}(k+1,\ n)$ is at least n when $k+1 \leq n$, we get $t_{\mathsf{Perm}}(k,\ n) = \Theta((n-k+1)t_{\mathsf{Perm}}(k+1,\ n))$ for $k\ <\ n$. Using the substitution method, we obtain $t_{\mathsf{Perm}}(1, n)\ =\ \Theta(n(n!))$, $n \geq 1$. □

Example 1.17 [Magic square] The next example we consider is a problem from recreational mathematics. A magic square is an $n \times n$ matrix of the integers 1 to n^2 such that the sum of every row, column, and diagonal is the same. Figure 1.2 gives an example magic square for the case $n = 5$. In this example, the common sum is 65.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Figure 1.2 Example magic square

H. Coxeter has given the following simple rule for generating a magic square when n is odd:

> Start with 1 in the middle of the top row; then go up and left, assigning numbers in increasing order to empty squares; if you fall off the square imagine the same square as tiling the plane and continue; if a square is occupied, move down instead and continue.

The magic square of Figure 1.2 was formed using this rule. Algorithm 1.15 is for creating an $n \times n$ magic square for the case in which n is odd. This results from Coxeter's rule.

The magic square is represented using a two-dimensional array having n rows and n columns. For this application it is convenient to number the rows (and columns) from 0 to $n-1$ rather than from 1 to n. Thus, when the algorithm "falls off the square," the **mod** operator sets i and/or j back to 0 or $n-1$.

The time to initialize and output the square is $\Theta(n^2)$. The third **for** loop (in which *key* ranges over 2 through n^2) is iterated $n^2 - 1$ times and each iteration takes $\Theta(1)$ time. So, this **for** loop takes $\Theta(n^2)$ time. Hence the overall time complexity of Magic is $\Theta(n^2)$. Since there are n^2 positions in which the algorithm must place a number, we see that $\Theta(n^2)$ is the best bound an algorithm for the magic square problem can have. □

Example 1.18 [Computing x^n] Our final example is to compute x^n for any real number x and integer $n \geq 0$. A naive algorithm for solving this problem is to perform $n-1$ multiplications as follows:

```
$power := x$;
for $i := 1$ to $n-1$ do $power := power * x$;
```

This algorithm takes $\Theta(n)$ time. A better approach is to employ the "repeated squaring" trick. Consider the special case in which n is an integral power of 2 (that is, in which n equals 2^k for some integer k). The following algorithm computes x^n.

```
$power := x$;
for $i := 1$ to $k$ do $power := power^2$;
```

The value of *power* after q iterations of the **for** loop is x^{2^q}. Therefore, this algorithm takes only $\Theta(k) = \Theta(\log n)$ time, which is a significant improvement over the run time of the first algorithm.

Can the same algorithm be used when n is not an integral power of 2? Fortunately, the answer is yes. Let $b_k b_{k-1} \cdots b_1 b_0$ be the binary representation of the integer n. This means that $n = \sum_{q=0}^{k} b_q 2^q$. Now,

$$x^n = x^{\sum_{q=0}^{k} b_q 2^q} = (x)^{b_0} * (x^2)^{b_1} * (x^4)^{b_2} * \cdots * (x^{2^k})^{b_k}$$

Also observe that b_0 is nothing but n **mod** 2 and that $\lfloor n/2 \rfloor$ is $b_k b_{k-1} \cdots b_1$ in binary form. These observations lead us to Exponentiate (Algorithm 1.16) for computing x^n.

```
Algorithm Magic($n$)
// Create a magic square of size $n$, $n$ being odd.
{
    if (($n$ mod 2) = 0) then
    {
        write ("$n$ is even"); return;
    }
    else
    {
        for $i := 0$ to $n - 1$ do  // Initialize $square$ to zero.
            for $j := 0$ to $n - 1$ do  $square[i, j] := 0$;
        $square[0, (n - 1)/2] := 1$; // Middle of first row
        // $(i, j)$ is the current position.
        $j := (n - 1)/2$;
        for $key := 2$ to $n^2$ do
        {
            // Move up and left. The next two if statements
            // may be replaced by the mod operator if
            // $-1$ mod $n$ has the value $n - 1$.
            if ($i \geq 1$) then $k := i - 1$; else $k := n - 1$;
            if ($j \geq 1$) then $l := j - 1$; else $l := n - 1$;
            if ($square[k, l] \geq 1$) then $i := (i + 1)$ mod $n$;
            else // $square[k, l]$ is empty.
            {
                $i := k$; $j := l$;
            }
            $square[i, j] := key$;
        }
        // Output the magic square.
        for $i := 0$ to $n - 1$ do
            for $j := 0$ to $n - 1$ do write ($square[i, j]$);
    }
}
```

Algorithm 1.15 Magic square

```
1   Algorithm Exponentiate($x, n$)
2   // Return $x^n$ for an integer $n \geq 0$.
3   {
4       $m := n$; $power := 1$; $z := x$;
5       while ($m > 0$) do
6       {
7           while (($m$ mod 2) = 0) do
8           {
9               $m := \lfloor m/2 \rfloor$; $z := z^2$;
10          }
11          $m := m - 1$; $power := power * z$;
12      }
13      return $power$;
14  }
```

Algorithm 1.16 Computation of x^n

Proving the correctness of this algorithm is left as an exercise. The variable m starts with the value of n, and after every iteration of the innermost **while** loop (line 7), its value decreases by a factor of at least 2. Thus there will be only $\Theta(\log n)$ iterations of the **while** loop of line 7. Each such iteration takes $\Theta(1)$ time. Whenever control exits from the innermost **while** loop, the value of m is odd and the instructions $m := m - 1$; $power := power * z$; are executed once. After this execution, since m becomes even, either the innermost **while** loop is entered again or the outermost **while** loop (line 5) is exited (in case $m = 0$). Therefore the instructions $m := m - 1$; $power := power * z$; can only be executed $O(\log n)$ times. In summary, the overall run time of Exponentiate is $\Theta(\log n)$. □

1.3.4 Practical Complexities

We have seen that the time complexity of an algorithm is generally some function of the instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change. The complexity function can also be used to compare two algorithms P and Q that perform the same task. Assume that algorithm P has complexity $\Theta(n)$ and algorithm Q has complexity $\Theta(n^2)$. We can assert that algorithm P is faster than algorithm Q for sufficiently large n. To see the validity of this assertion, observe that the computing time of P is bounded from above by cn for some constant c and for all n, $n \geq n_1$, whereas that of Q is bounded from below by dn^2 for some constant d and all n, $n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, algorithm P is faster than algorithm Q whenever $n \geq \max\{n_1, n_2, c/d\}$.

You should always be cautiously aware of the presence of the phrase "sufficiently large" in an assertion like that of the preceding discussion. When deciding which of the two algorithms to use, you must know whether the n you are dealing with is, in fact, sufficiently large. If algorithm P runs in $10^6 n$ milliseconds, whereas algorithm Q runs in n^2 milliseconds, and if you always have $n \leq 10^6$, then, other factors being equal, algorithm Q is the one to use.

To get a feel for how the various functions grow with n, you are advised to study Table 1.7 and Figure 1.3 very closely. It is evident from Table 1.7 and Figure 1.3 that the function 2^n grows very rapidly with n. In fact, if an algorithm needs 2^n steps for execution, then when $n = 40$, the number of steps needed is approximately $1.1 * 10^{12}$. On a computer performing one billion steps per second, this would require about 18.3 minutes. If $n = 50$, the same algorithm would run for about 13 days on this computer. When $n = 60$, about 310.56 years are required to execute the algorithm and when $n = 100$, about $4 * 10^{13}$ years are needed. So, we may conclude that the utility of algorithms with exponential complexity is limited to small n (typically $n \leq 40$).

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Table 1.7 Function values

Algorithms that have a complexity that is a polynomial of high degree are also of limited utility. For example, if an algorithm needs n^{10} steps, then using our 1-billion-steps-per-second computer, we need 10 seconds when $n = 10$, 3171 years when $n = 100$, and $3.17 * 10^{13}$ years when $n = 1000$. If the algorithm's complexity had been n^3 steps instead, then we would need one second when $n = 1000$, 110.67 minutes when $n = 10{,}000$, and 11.57 days when $n = 100{,}000$.

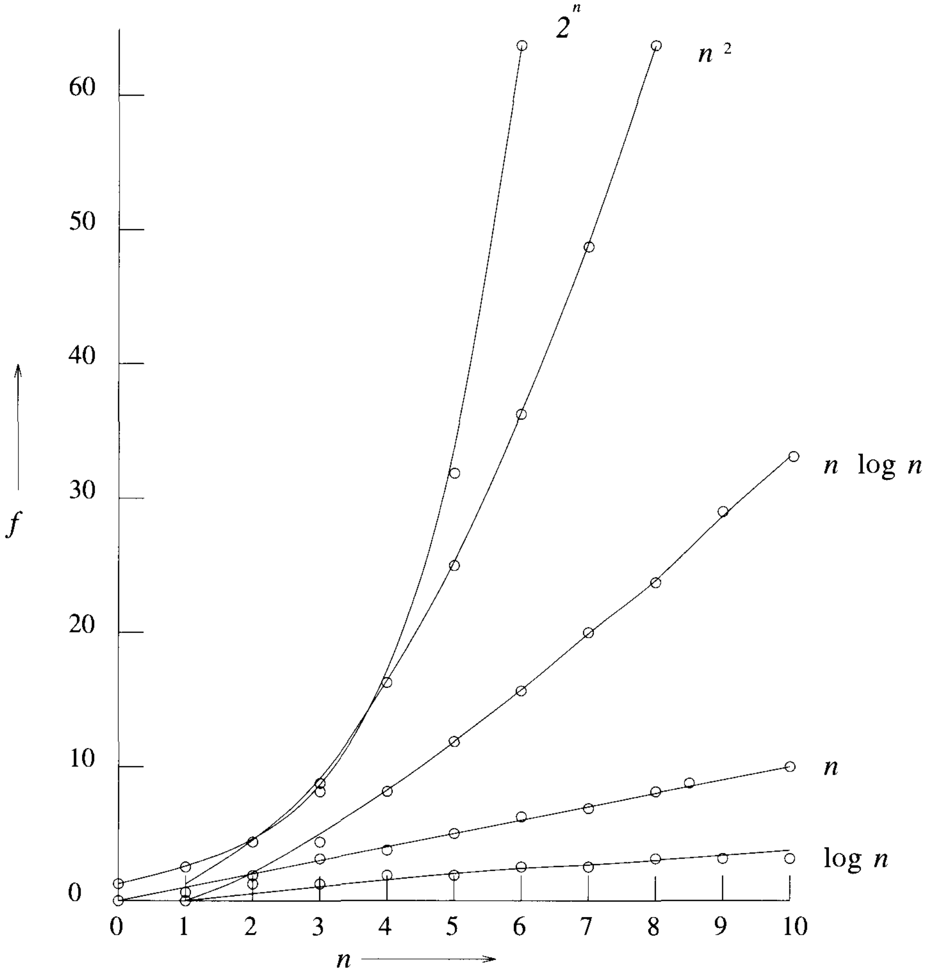

Figure 1.3 Plot of function values

Table 1.8 gives the time needed by a one-billion-steps-per-second computer to execute an algorithm of complexity $f(n)$ instructions. You should note that currently only the fastest computers can execute about 1 billion instructions per second. From a practical standpoint, it is evident that for reasonably large n (say $n > 100$), only algorithms of small complexity (such as n, $n \log n$, n^2, and n^3) are feasible. Further, this is the case even if you could build a computer capable of executing 10^{12} instructions per second. In this case, the computing times of Table 1.8 would decrease by a factor of 1000. Now, when $n = 100$, it would take 3.17 years to execute n^{10} instructions and $4 * 10^{10}$ years to execute 2^n instructions.

	Time for $f(n)$ instructions on a 10^9instr/sec computer						
n	$f(n) = n$	$f(n) = n \log_2 n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = n^4$	$f(n) = n^{10}$	$f(n) = 2^n$
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 hr	1 ms
30	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121.36 d	18.3 min
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 yr	13 d
100	.1 μs	.66 μs	10 μs	1 ms	100 ms	3171 yr	$4*10^{13}$ yr
1,000	1 μs	9.96 μs	1 ms	1 s	16.67 min	$3.17*10^{13}$ yr	$32*10^{283}$ yr
10,000	10 μs	130 μs	100 ms	16.67 min	115.7 d	$3.17*10^{23}$ yr	
100,000	100 μs	1.66 ms	10 s	11.57 d	3171 yr	$3.17*10^{33}$ yr	
1,000,000	1 ms	19.92 ms	16.67 min	31.71 yr	$3.17*10^7$ yr	$3.17*10^{43}$ yr	

Table 1.8 Times on a 1-billion-steps-per-second computer

1.3.5 Performance Measurement

Performance measurement is concerned with obtaining the space and time requirements of a particular algorithm. These quantities depend on the compiler and options used as well as on the computer on which the algorithm is run. Unless otherwise stated, all performance values provided in this book are obtained using the Gnu C++ compiler, the default compiler options, and the Sparc 10/30 computer workstation.

In keeping with the discussion of the preceding section, we do not concern ourselves with the space and time needed for compilation. We justify this by the assumption that each program (after it has been fully debugged) is compiled once and then executed several times. Certainly, the space and time needed for compilation are important during program testing, when more time is spent on this task than in running the compiled code.

We do not consider measuring the run-time space requirements of a program. Rather, we focus on measuring the computing time of a program. To obtain the computing (or run) time of a program, we need a clocking procedure. We assume the existence of a program GetTime() that returns the current time in milliseconds.

Suppose we wish to measure the worst-case performance of the sequential search algorithm (Algorithm 1.17). Before we can do this, we need to (1) decide on the values of n for which the times are to be obtained and (2) determine, for each of the above values of n, the data that exhibit the worst-case behavior.

```
Algorithm SeqSearch($a, x, n$)
// Search for $x$ in $a[1:n]$. $a[0]$ is used as additional space.
{
    $i := n$; $a[0] := x$;
    while ($a[i] \neq x$) do $i := i - 1$;
    return $i$;
}
```

Algorithm 1.17 Sequential search

The decision on which values of n to use is based on the amount of timing we wish to perform and also on what we expect to do with the times once they are obtained. Assume that for Algorithm 1.17, our intent is simply to predict how long it will take, in the worst case, to search for x, given the size n of a. An asymptotic analysis reveals that this time is $\Theta(n)$. So, we expect a plot of the times to be a straight line. Theoretically, if we know the times for any two values of n, the straight line is determined, and we can obtain the time for all other values of n from this line. In practice, we need the times for more than two values of n. This is so for the following reasons:

1. Asymptotic analysis tells us the behavior only for sufficiently large values of n. For smaller values of n, the run time may not follow the asymptotic curve. To determine the point beyond which the asymptotic curve is followed, we need to examine the times for several values of n.

2. Even in the region where the asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve (straight line in the case of Algorithm 1.17) because of the effects of low-order terms that are discarded in the asymptotic analysis. For instance, an algorithm with asymptotic complexity $\Theta(n)$ can have time complexity $c_1 n + c_2 \log n + c_3$ or, for that matter, any other function of n in which the highest-order term is $c_1 n$ for some constant c_1, $c_1 > 0$.

It is reasonable to expect that the asymptotic behavior of Algorithm 1.17 begins for some n that is smaller than 100. So, for $n > 100$, we obtain the run time for just a few values. A reasonable choice is $n = 200$, 300, 400, ... , 1000. There is nothing magical about this choice of values. We can just as well use $n = 500, 1,000, 1,500, \ldots, 10,000$ or $n = 512$, 1,024, 2,048,..., 2^{15}. It costs us more in terms of computer time to use the latter choices, and we probably do not get any better information about the run time of Algorithm 1.17 using these choices.

For n in the range [0, 100] we carry out a more-refined measurement, since we are not quite sure where the asymptotic behavior begins. Of course, if our measurements show that the straight-line behavior does not begin in this range, we have to perform a more-detailed measurement in the range [100, 200], and so on, until the onset of this behavior is detected. Times in the range [0, 100] are obtained in steps of 10 beginning at $n = 0$.

Algorithm 1.17 exhibits its worst-case behavior when x is chosen such that it is not one of the $a[i]$'s. For definiteness, we set $a[i] = i$, $1 \le i \le n$, and $x = 0$. At this time, we envision using an algorithm such as Algorithm 1.18 to obtain the worst-case times.

```
Algorithm TimeSearch()
{
    for $j := 1$ to 1000 do  $a[j] := j$;
    for $j := 1$ to 10 do
    {
        $n[j] := 10 * (j - 1)$; $n[j + 10] := 100 * j$;
    }
    for $j := 1$ to 20 do
    {
        $h :=$ GetTime();
        $k :=$ SeqSearch($a, 0, n[j]$);
        $h1 :=$ GetTime();
        $t := h1 - h$;
        write ($n[j]$, $t$);
    }
}
```

Algorithm 1.18 Algorithm to time Algorithm 1.17

The timing results of this algorithm is summarized in Table 1.9. The times obtained are too small to be of any use to us. Most of the times are zero; this indicates that the precision of our clock is inadequate. The nonzero times are just noise and are not representative of the time taken.

n	time	n	time
0	0	100	0
10	0	200	0
20	0	300	1
30	0	400	0
40	0	500	1
50	0	600	0
60	0	700	0
70	0	800	1
80	0	900	0
90	0	1000	0

Table 1.9 Timing results of Algorithm 1.18. Times are in milliseconds.

> *To time a short event, it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.*

Since our clock has an accuracy of about one-tenth of a second, we should not attempt to time any single event that takes less than about one second. With an event time of at least ten seconds, we can expect our observed times to be accurate to one percent.

The body of Algorithm 1.18 needs to be changed to that of Algorithm 1.19. In this algorithm, $r[i]$ is the number of times the search is to be repeated when the number of elements in the array is $n[i]$. Notice that rearranging the timing statements as in Algorithm 1.20 or 1.21 does not produce the desired results. For instance, from the data of Table 1.9, we expect that with the structure of Algorithm 1.20, the value output for $n = 0$ will still be 0. This is because there is a chance that in every iteration of the **for** loop, the clock does not change between the two times GetTime() is called. With the structure of Algorithm 1.21, we expect the algorithm never to exit the **while** loop when $n = 0$ (in reality, the loop will be exited because occasionally the measured time will turn out to be a few milliseconds).

Yet another alternative is shown in Algorithm 1.22. This approach can be expected to yield satisfactory times. It cannot be used when the timing procedure available gives us only the time since the last invocation of GetTime. Another difficulty is that the measured time includes the time needed to read the clock. For small n, this time may be larger than the time to run SeqSearch. This difficulty can be overcome by determining the time taken by the timing procedure and subtracting this time later.

```
Algorithm TimeSearch()
{
    // Repetition factors
    $r[21] := \{0, 200000, 200000, 150000, 100000, 100000, 100000,$
        $50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,$
        $50000, 50000, 25000, 25000, 25000, 25000\};$
    for $j := 1$ to 1000 do $a[j] := j;$
    for $j := 1$ to 10 do
    {
        $n[j] := 10 * (j - 1); n[j + 10] := 100 * j;$
    }
    for $j := 1$ to 20 do
    {
        $h :=$ GetTime();
        for $i := 1$ to $r[j]$ do $k :=$ SeqSearch$(a, 0, n[j]);$
        $h1 :=$ GetTime();
        $t1 := h1 - h;$
        $t := t1; t := t/r[j];$
        write $(n[j], t1, t);$
    }
}
```

Algorithm 1.19 Timing algorithm

```
$t := 0;$
for $i := 1$ to $r[j]$ do
{
    $h :=$ GetTime();
    $k :=$ SeqSearch$(a, 0, n[j]);$
    $h1 :=$ GetTime();
    $t := t + h1 - h;$
}
$t := t/r[j];$
```

Algorithm 1.20 Improper timing construct

```
$t := 0$;
while ($t < DESIRED\_TIME$) do
{
    $h :=$ GetTime();
    $k :=$ SeqSearch($a, 0, n[j]$);
    $h1 :=$ GetTime();
    $t := t + h1 - h$;
}
```

Algorithm 1.21 Another improper timing construct

```
$h :=$ GetTime(); $t := 0$;
while ($t < DESIRED\_TIME$) do
{
    $k :=$ SeqSearch($a, 0, n[j]$);
    $h1 :=$ GetTime();
    $t := h1 - h$;
}
```

Algorithm 1.22 An alternate timing construct

Timing results of Algorithm 1.19, is given in Table 1.10. The times for n in the range [0, 1000] are plotted in Figure 1.4. Values in the range [10, 100] have not been plotted. The linear dependence of the worst-case time on n is apparent from this graph.

n	$t1$	t	n	$t1$	t
0	308	0.002	100	1683	0.034
10	923	0.005	200	3359	0.067
20	1181	0.008	300	4693	0.094
30	1087	0.011	400	6323	0.126
40	1384	0.014	500	7799	0.156
50	1691	0.017	600	9310	0.186
60	999	0.020	700	5419	0.217
70	1156	0.023	800	6201	0.248
80	1306	0.026	900	6994	0.280
90	1460	0.029	1000	7725	0.309

Times are in milliseconds

Table 1.10 Worst-case run times for Algorithm 1.17

The graph of Figure 1.4 can be used to predict the run time for other values of n. We can go one step further and get the equation of the straight line. The equation of this line is $t = c + mn$, where m is the slope and c the value for $n = 0$. From the graph, we see that $c = 0.002$. Using the point $n = 600$ and $t = 0.186$, we obtain $m = (t - c)/n = 0.184/600 = 0.0003067$. So the line of Figure 1.4 has the equation $t = 0.002 + 0.0003067n$, where t is the time in milliseconds. From this, we expect that when $n = 1000$, the worst-case search time will be 0.3087 millisecond, and when $n = 500$, it will be 0.155 millisecond. Compared to the observed times of Table 1.10, we see that these figures are very accurate!

Summary of Running Time Calculation

To obtain the run time of a program, we need to plan the experiment. The following issues need to be addressed during the planning stage:

1. What is the accuracy of the clock? How accurate do our results have to be? Once the desired accuracy is known, we can determine the length of the shortest event that should be timed.

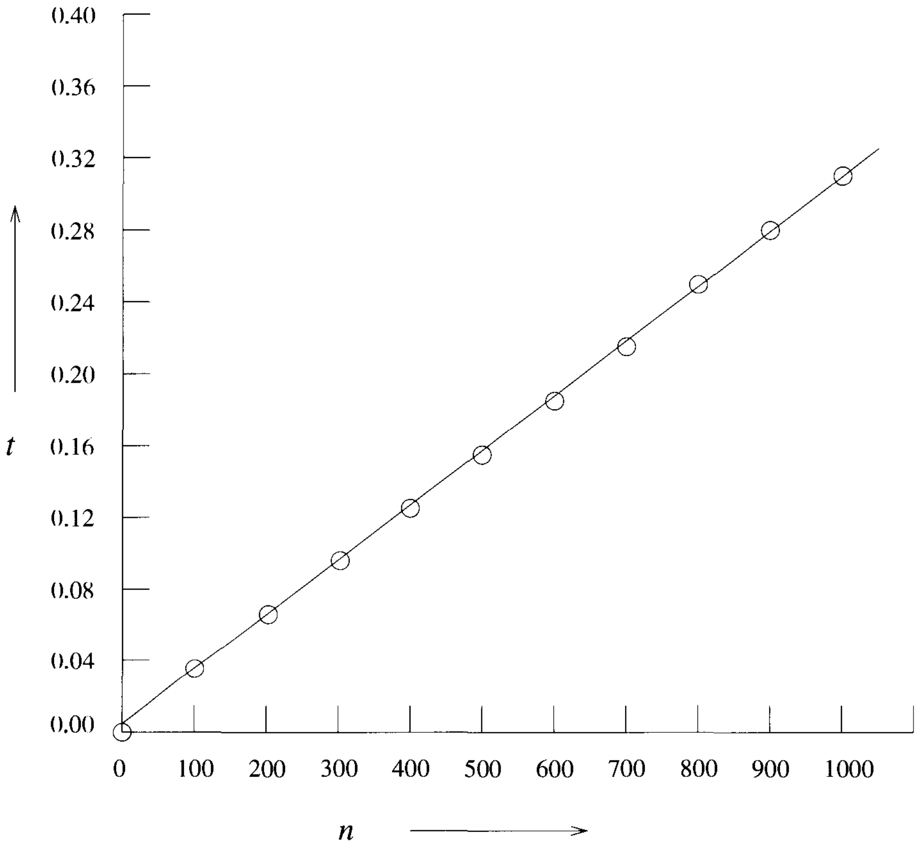

Figure 1.4 Plot of the data in Table 1.10

2. For each instance size, a repetition factor needs to be determined. This is to be chosen such that the event time is at least the minimum time that can be clocked with the desired accuracy.

3. Are we measuring worst-case or average performance? Suitable test data need to be generated.

4. What is the purpose of the experiment? Are the times being obtained for comparative purposes, or are they to be used to predict run times? If the latter is the case, then contributions to the run time from such sources as the repetition loop and data generation need to be subtracted (in case they are included in the measured time). If the former is the case, then these times need not be subtracted (provided they are the same for all programs being compared).

5. In case the times are to be used to predict run times, then we need to fit a curve through the points. For this, the asymptotic complexity should be known. If the asymptotic complexity is linear, then a least-squares straight line can be fit; if it is quadratic, then a parabola can be used (that is, $t = a_0 + a_1 n + a_2 n^2$). If the complexity is $\Theta(n \log n)$, then a least-squares curve of the form $t = a_0 + a_1 n + a_2 n \log_2 n$ can be fit. When obtaining the least-squares approximation, one should discard data corresponding to small values of n, since the program does not exhibit its asymptotic behavior for these n.

Generating Test Data

Generating a data set that results in the worst-case performance of an algorithm is not always easy. In some cases, it is necessary to use a computer program to generate the worst-case data. In other cases, even this is very difficult. In these cases, another approach to estimating worst-case performance is taken. For each set of values of the instance characteristics of interest, we generate a suitably large number of random test data. The run times for each of these test data are obtained. The maximum of these times is used as an estimate of the worst-case time for this set of values of the instance characteristics.

To measure average-case times, it is usually not possible to average over all possible instances of a given characteristic. Although it is possible to do this for sequential search, it is not possible for a sort algorithm. If we assume that all keys are distinct, then for any given n, $n!$ different permutations need to be used to obtain the average time. Obtaining average-case data is usually much harder than obtaining worst-case data. So, we often adopt the strategy outlined above and simply obtain an estimate of the average time on a suitable set of test data.

Whether we are estimating worst-case or average time using random data, the number of instances that we can try is generally much smaller than the total number of such instances. Hence, it is desirable to analyze the algorithm being tested to determine classes of data that should be generated for the experiment. This is a very algorithm-specific task, and we do not go into it here.

EXERCISES

1. Compare the two functions n^2 and $2^n/4$ for various values of n. Determine when the second becomes larger than the first.

2. Prove by induction:

   (a) $\sum_{i=1}^{n} i \;=\; n(n+1)/2,\; n \geq 1$

   (b) $\sum_{i=1}^{n} i^2 \;=\; n(n+1)(2n+1)/6,\; n \geq 1$

   (c) $\sum_{i=0}^{n} x^i \;=\; (x^{n+1}-1)/(x-1),\; x \neq 1,\; n \geq 0$

3. Determine the frequency counts for all statements in the following two algorithm segments:

```
for $i := 1$ to $n$ do
    for $j := 1$ to $i$ do
        for $k := 1$ to $j$ do
            $x := x + 1;$
```

(a)

```
$i := 1;$
while $(i \leq n)$ do
{
    $x := x + 1;$
    $i := i + 1;$
}
```

(b)

4. (a) Introduce statements to increment *count* at all appropriate points in Algorithm 1.23.

   (b) Simplify the resulting algorithm by eliminating statements. The simplified algorithm should compute the same value for *count* as computed by the algorithm of part (a).

   (c) What is the exact value of *count* when the algorithm terminates? You may assume that the initial value of *count* is 0.

   (d) Obtain the step count for Algorithm 1.23 using the frequency method. Clearly show the step count table.

5. Do Exercise 4 for Transpose (Algorithm 1.24).

6. Do Exercise 4 for Algorithm 1.25. This algorithm multiplies two $n \times n$ matrices a and b.

```
Algorithm D($x, n$)
{
    $i := 1$;
    repeat
    {
        $x[i] := x[i] + 2$; $i := i + 2$;
    } until ($i > n$);
    $i := 1$;
    while ($i \leq \lfloor n/2 \rfloor$) do
    {
        $x[i] := x[i] + x[i+1]$; $i := i + 1$;
    }
}
```

Algorithm 1.23 Example algorithm

```
Algorithm Transpose($a, n$)
{
    for $i := 1$ to $n - 1$ do
        for $j := i + 1$ to $n$ do
        {
            $t := a[i, j]$; $a[i, j] := a[j, i]$; $a[j, i] := t$;
        }
}
```

Algorithm 1.24 Matrix transpose

```
Algorithm Mult($a, b, c, n$)
{
    for $i := 1$ to $n$ do
        for $j := 1$ to $n$ do
        {
            $c[i, j] := 0$;
            for $k := 1$ to $n$ do
                $c[i, j] := c[i, j] + a[i, k] * b[k, j]$;
        }
}
```

Algorithm 1.25 Matrix multiplication

7. (a) Do Exercise 4 for Algorithm 1.26. This algorithm multiplies two matrices a and b, where a is an $m \times n$ matrix and b is an $n \times p$ matrix.

```
Algorithm Mult($a, b, c, m, n, p$)
{
    for $i := 1$ to $m$ do
        for $j := 1$ to $p$ do
        {
            $c[i, j] := 0$;
            for $k := 1$ to $n$ do
                $c[i, j] := c[i, j] + a[i, k] * b[k, j]$;
        }
}
```

Algorithm 1.26 Matrix multiplication

(b) Under what conditions is it profitable to interchange the two outermost **for** loops?

8. Show that the following equalities are correct:

(a) $5n^2 - 6n \;=\; \Theta(n^2)$

(b) $n! \;=\; O(n^n)$

(c) $2n^2 2^n + n \log n \;=\; \Theta(n^2 2^n)$

(d) $\sum_{i=0}^{n} i^2 \;=\; \Theta(n^3)$

(e) $\sum_{i=0}^{n} i^3 = \Theta(n^4)$.

(f) $n^{2^n} + 6 * 2^n = \Theta(n^{2^n})$

(g) $n^3 + 10^6 n^2 = \Theta(n^3)$

(h) $6n^3/(\log n + 1) = O(n^3)$

(i) $n^{1.001} + n \log n = \Theta(n^{1.001})$

(j) $n^{k+\epsilon} + n^k \log n = \Theta(n^{k+\epsilon})$ for all fixed k and ϵ, $k \geq 0$ and $\epsilon > 0$

(k) $10n^3 + 15n^4 + 100n^2 2^n = O(100n^2 2^n)$

(l) $33n^3 + 4n^2 = \Omega(n^2)$

(m) $33n^3 + 4n^2 = \Omega(n^3)$

9. Show that the following equalities are incorrect:

   (a) $10n^2 + 9 = O(n)$

   (b) $n^2 \log n = \Theta(n^2)$

   (c) $n^2/\log n = \Theta(n^2)$

   (d) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$

10. Prove Theorems 1.3 and 1.4.

11. Analyze the computing time of SelectionSort (Algorithm 1.2).

12. Obtain worst-case run times for SelectionSort (Algorithm 1.2). Do this for suitable values of n in the range [0, 100]. Your report must include a plan for the experiment as well as the measured times. These times are to be provided both in a table and as a graph.

13. Consider the algorithm Add (Algorithm 1.11).

   (a) Obtain run times for $n = 1, 10, 20, \ldots, 100$.

   (b) Plot the times obtained in part (a).

14. Do the previous exercise for matrix multiplication (Algorithm 1.26).

15. A complex-valued matrix X is represented by a pair of matrices (A, B), where A and B contain real values. Write an algorithm that computes the product of two complex-valued matrices (A, B) and (C, D), where $(A, B) * (C, D) = (A + iB) * (C + iD) = (AC - BD) + i(AD + BC)$. Determine the number of additions and multiplications if the matrices are all $n \times n$.

1.4 RANDOMIZED ALGORITHMS

1.4.1 Basics of Probability Theory

Probability theory has the goal of characterizing the outcomes of natural or conceptual "experiments." Examples of such experiments include tossing a coin ten times, rolling a die three times, playing a lottery, gambling, picking a ball from an urn containing white and red balls, and so on.

Each possible outcome of an experiment is called a *sample point* and the set of all possible outcomes is known as the *sample space* S. In this text we assume that S is finite (such a sample space is called a *discrete sample space*). An *event* E is a subset of the sample space S. If the sample space consists of n sample points, then there are 2^n possible events.

Example 1.19 [Tossing three coins] When a coin is tossed, there are two possible outcomes: heads (H) and tails (T). Consider the experiment of throwing three coins. There are eight possible outcomes: HHH, HHT, HTH, HTT, THH, THT, TTH, and TTT. Each such outcome is a sample point. The sets $\{HHT, HTT, TTT\}$, $\{HHH, TTT\}$, and $\{\ \}$ are three possible events. The third event has no sample points and is the empty set. For this experiment there are 2^8 possible events. □

Definition 1.9 [Probability] The probability of an event E is defined to be $\frac{|E|}{|S|}$, where S is the sample space. □

Example 1.20 [Tossing three coins] The probability of the event $\{HHT, HTT, TTT\}$ is $\frac{3}{8}$. The probability of the event $\{HHH, TTT\}$ is $\frac{2}{8}$ and that of the event $\{\ \}$ is zero. □

Note that the probability of S, the sample space, is 1.

Example 1.21 [Rolling two dice] Let us look at the experiment of rolling two (six-faced) dice. There are 36 possible outcomes some of which are $(1,1), (1,2), (1,3), \ldots$. What is the probability that the sum of the two faces is 10? The event that the sum is 10 consists of the following sample points: $(1,9), (2,8), (3,7), (4,6), (5,5), (6,4), (7,3), (8,2)$, and $(9,1)$. Therefore, the probability of this event is $\frac{9}{36} = \frac{1}{4}$. □

Definition 1.10 [Mutual exclusion] Two events E_1 and E_2 are said to be *mutually exclusive* if they do not have any common sample points, that is, if $E_1 \cap E_2 = \Phi$. □

Example 1.22 [Tossing three coins] When we toss three coins, let E_1 be the event that there are two H's and let E_2 be the event that there are at least two T's. These two events are mutually exclusive since there are no common sample points. On the other hand, if E_2' is defined to be the event that there is at least one T, then E_1 and E_2' will *not* be mutually exclusive since they will have THH, HTH, and HHT as common sample points. □

The probability of event E is denoted as $Prob.[E]$. The *complement* of E, denoted $\bar{E}$, is defined to be $S - E$. If E_1 and E_2 are two events, the probability of E_1 or E_2 or both happening is denoted as $Prob.[E_1 \cup E_2]$. The probability of both E_1 and E_2 occurring at the same time is denoted as $Prob.[E_1 \cap E_2]$. The corresponding event is $E_1 \cap E_2$.

Theorem 1.5

$$\begin{array}{rrcl} 1. & Prob.[\bar{E}] & = & 1 - Prob.[E]. \\ 2. & Prob.[E_1 \cup E_2] & = & Prob.[E_1] + Prob.[E_2] - Prob.[E_1 \cap E_2] \\ & & \leq & Prob.[E_1] + Prob.[E_2] \end{array}$$

Definition 1.11 [Conditional probability] Let E_1 and E_2 be any two events of an experiment. The *conditional probability of E_1 given E_2*, denoted by $Prob.\ [E_1|E_2]$, is defined as $\frac{Prob.[E_1 \cap E_2]}{Prob.[E_2]}$. □

Example 1.23 [Tossing four coins] Consider the experiment of tossing four coins. Let E_1 be the event that the number of H's is even and let E_2 be the event that there is at least one H. Then, E_2 is the complement of the event that there are no H's. The probability of no H's is $\frac{1}{16}$. Therefore, $Prob.[E_2] = 1 - \frac{1}{16} = \frac{15}{16}$. $Prob.[E_1 \cap E_2]$ is $\frac{7}{16}$ since the event $E_1 \cap E_2$ has the seven sample points $HHHH$, $HHTT$, $HTHT$, $HTTH$, $THHT$, $THTH$, and $TTHH$. Thus, $Prob.[E_1|E_2]$ is $\frac{7/16}{15/16} = \frac{7}{15}$. □

Definition 1.12 [Independence] Two events E_1 and E_2 are said to be *independent* if $Prob.[E_1 \cap E_2] = Prob.[E_1] * Prob.[E_2]$. □

Example 1.24 [Rolling a die twice] Intuitively, we say two events E_1 and E_2 are independent if the probability of one event happening is in no way affected by the occurrence of the other event. In other words, if $Prob.[E_1|E_2] = Prob.[E_1]$, these two events are independent. Suppose we roll a die twice. What is the probability that the outcome of the second roll is 5 (call this event E_1), given that the outcome of the first roll is 4 (call this event E_2)? The answer is $\frac{1}{6}$ no matter what the outcome of the first roll is. In this case E_1 and E_2 are independent. Therefore, $Prob.[E_1 \cap E_2] = \frac{1}{6} * \frac{1}{6} = \frac{1}{36}$. □

Example 1.25 [Flipping a coin 100 times] If a coin is flipped 100 times what is the probability that all of the outcomes are tails? The probability that the first outcome is T is $\frac{1}{2}$. Since the outcome of the second flip is independent of the outcome of the first flip, the probability that the first two outcomes are T's can be obtained by multiplying the corresponding probabilities to get $\frac{1}{4}$. Extending the argument to all 100 outcomes, we conclude that the probability of obtaining 100 T's is $\left(\frac{1}{2}\right)^{100}$. In this case we say the outcomes of the 100 coin flips are *mutually independent.* □

Definition 1.13 [Random variable] Let S be the sample space of an experiment. A *random variable* on S is a function that maps the elements of S to the set of real numbers. For any sample point $s \in S$, $X(s)$ denotes the image of s under this mapping. If the range of X, that is, the set of values X can take, is finite, we say X is *discrete.*

Let the range of a discrete random variable X be $\{r_1, r_2, \ldots, r_m\}$. Then, $Prob.[X = r_i]$, for any i, is defined to be the the number of sample points whose image is r_i divided by the number of sample points in S. In this text we are concerned mostly with discrete random variables. □

Example 1.26 We flip a coin four times. The sample space consists of 2^4 sample points. We can define a random variable X on S as the number of heads in the coin flips. For this random variable, then, $X(HTHH) = 3$, $X(HHHH) = 4$, and so on. The possible values that X can take are $0, 1, 2, 3$, and 4. Thus X is discrete. $Prob.[X = 0]$ is $\frac{1}{16}$, since the only sample point whose image is 0 is $TTTT$. $Prob.[X = 1]$ is $\frac{4}{16}$, since the four sample points $HTTT$, $THTT$, $TTHT$, and $TTTH$ have 1 as their image. □

Definition 1.14 [Expected value] If the sample space of an experiment is $S = \{s_1, s_2, \ldots, s_n\}$, the *expected value* or the *mean* of any random variable X is defined to be $\sum_{i=1}^{n} Prob.[s_i] * X(s_i) = \frac{1}{n} \sum_{i=1}^{n} X(s_i)$. □

Example 1.27 [Coin tosses] The sample space corresponding to the experiment of tossing three coins is $S = \{HHH, HHT, HTH, HTT, THH, THT, TTH, TTT\}$. If X is the number of heads in the coin flips, then the expected value of X is $\frac{1}{8}(3 + 2 + 2 + 1 + 2 + 1 + 1 + 0) = 1.5$. □

Definition 1.15 [Probability distribution] Let X be a discrete random variable defined over the sample space S. Let $\{r_1, r_2, \ldots, r_m\}$ be its range. Then, the *probability distribution* of X is the sequence $Prob.[X = r_1]$, $Prob.[X = r_2]$, ... , $Prob.[X = r_m]$. Notice that $\sum_{i=1}^{m} Prob.[X = r_i] = 1$. □

Example 1.28 [Coin tosses] If a coin is flipped three times and X is the number of heads, then X can take on four values, 0, 1, 2, and 3. The probability distribution of X is given by $Prob.[X = 0] = \frac{1}{8}$, $Prob.[X = 1] = \frac{3}{8}$, $Prob.[X = 2] = \frac{3}{8}$, and $Prob.[X = 3] = \frac{1}{8}$. □

Definition 1.16 [Binomial distribution] A *Bernoulli* trial is an experiment that has two possible outcomes, namely, *success* and *failure*. The probability of success is p. Consider the experiment of conducting the Bernoulli trial n times. This experiment has a sample space S with 2^n sample points. Let X be a random variable on S defined to be the numbers of successes in the n trials. The variable X is said to have a *binomial distribution* with parameters $(n,\ p)$. The expected value of X is np. Also,

$$Prob.[X = i] \;=\; \binom{n}{i} p^i (1-p)^{n-i}$$

□

In several applications, it is necessary to estimate the probabilities at the tail ends of probability distributions. One such estimate is provided by the following lemma.

Lemma 1.1 [Markov's inequality] If X is any nonnegative random variable whose mean is μ, then

$$Prob.[X \geq x] \;\leq\; \frac{\mu}{x}$$

□

Example 1.29 Let μ be the mean of a random variable X. We can use Markov's lemma (also called Markov's inequality) to make the following statement: "The probability that the value of X exceeds 2μ is $\leq \frac{1}{2}$." Consider the example: if we toss a coin 1000 times, what is the probability that the number of heads is $\geq$ 600? If X is the number of heads in 1000 tosses, then, the expected value of X, $E[X]$, is 500. Applying Markov's inequality with $x = 600$ and $\mu = 500$, we infer that $P[X \geq 600] \leq \frac{5}{6}$. □

Though Markov's inequality can be applied to any nonnegative random variable, it is rather weak. We can obtain tighter bounds for a number of important distributions including the binomial distribution. These bounds are due to Chernoff. Chernoff bounds as applied to the binomial distribution are employed in this text to analyze randomized algorithms.

Lemma 1.2 [Chernoff bounds] If X is a binomial with parameters $(n,\ p)$, and $m > np$ is an integer, then

$$Prob.(X \geq m) \ \leq \ \left(\frac{np}{m}\right)^m e^{(m-np)}. \tag{1.1}$$

$$\text{Also,} \ \ Prob.(X \leq \lfloor(1-\epsilon)pn\rfloor) \ \leq \ e^{(-\epsilon^2 np/2)} \tag{1.2}$$

$$\text{and} \ \ Prob.(X \geq \lceil(1+\epsilon)np\rceil) \ \leq \ e^{(-\epsilon^2 np/3)} \tag{1.3}$$

for all $0 < \epsilon < 1$. □

Example 1.30 Consider the experiment of tossing a coin 1000 times. We want to determine the probability that the number X of heads is ≥ 600. We can use Equation 1.3 to estimate this probability. The value for ϵ here is 0.2. Also, $n = 1000$ and $p = \frac{1}{2}$. Equation 1.3 now becomes

$$P[X \geq 600] \leq e^{[-(0.2)^2(500/3)]} = e^{-20/3} \leq 0.001273$$

This estimate is more precise than that given by Markov's inequality. □

1.4.2 Randomized Algorithms: An Informal Description

A randomized algorithm is one that makes use of a randomizer (such as a random number generator). Some of the decisions made in the algorithm depend on the output of the randomizer. Since the output of any randomizer might differ in an unpredictable way from run to run, the output of a randomized algorithm could also differ from run to run for the same input. The execution time of a randomized algorithm could also vary from run to run for the same input.

Randomized algorithms can be categorized into two classes: The first is algorithms that always produce the same (correct) output for the same input. These are called *Las Vegas* algorithms. The execution time of a Las Vegas algorithm depends on the output of the randomizer. If we are lucky, the algorithm might terminate fast, and if not, it might run for a longer period of time. In general the execution time of a Las Vegas algorithm is characterized as a random variable (see Section 1.4.1 for a definition). The second is algorithms whose outputs might differ from run to run (for the same input). These are called *Monte Carlo* algorithms. Consider any problem for which there are only two possible answers, say, yes and no. If a Monte Carlo algorithm is employed to solve such a problem, then the algorithm might give incorrect answers depending on the output of the randomizer. We require that the probability of an incorrect answer from a Monte Carlo algorithm be low. Typically, for a fixed input, a Monte Carlo algorithm does not display much variation in execution time between runs, whereas in the case of a Las Vegas algorithm this variation is significant.

We can think of a randomized algorithm with one possible randomizer output to be different from the same algorithm with a different possible randomizer output. Therefore, a randomized algorithm can be viewed as a family of algorithms. For a given input, some of the algorithms in this family may run for indefinitely long periods of time (or may give incorrect answers). The objective in the design of a randomized algorithm is to ensure that the number of such bad algorithms in the family is only a small fraction of the total number of algorithms. If for *any* input we can show that at least $1 - \epsilon$ (ϵ being very close to 0) fraction of algorithms in the family will run quickly (respectively give the correct answer) on that input, then clearly, a random algorithm in the family will run quickly (or output the correct answer) on any input with probability $\geq 1 - \epsilon$. In this case we say that this family of algorithms (or this randomized algorithm) runs quickly (respectively gives the correct answer) with probability at least $1 - \epsilon$, where ϵ is called the *error probability*.

Definition 1.17 [The $\widetilde{O}()$] Just like the $O()$ notation is used to characterize the run times of non randomized algorithms, $\widetilde{O}()$ is used for characterizing the run times of Las Vegas algorithms. We say a Las Vegas algorithm has a resource (time, space, and so on.) bound of $\widetilde{O}(g(n))$ if there exists a constant c such that the amount of resource used by the algorithm (on any input of size n) is no more than $c\alpha g(n)$ with probability $\geq 1 - \frac{1}{n^\alpha}$. We shall refer to these bounds as *high probability* bounds.

Similar definitions apply also to such functions as $\widetilde{\Theta}()$, $\widetilde{\Omega}()$, $\widetilde{o}()$, etc. □

Definition 1.18 [High probability] By *high probability* we mean a probability of $\geq 1 - n^{-\alpha}$ for any fixed α. We call α the probability parameter. □

As mentioned above, the run time T of any Las Vegas algorithm is typically characterized as a random variable over a sample space S. The sample points of S are all possible outcomes for the randomizer used in the algorithm. Though it is desirable to obtain the distribution of T, often this is a challenging and unnecessary task. The expected value of T often suffices as a good indicator of the run time. We can do better than obtaining the mean of T but short of computing the exact distribution by obtaining the high probability bounds. The high probability bounds of our interest are of the form "With high probability the value of T will not exceed T_0," for some appropriate T_0.

Several results from probability theory can be employed to obtain high probability bounds on any random variable. Two of the more useful such results are Markov's inequality and Chernoff bounds.

Next we give two examples of randomized algorithms. The first is of the Las Vegas type and the second is of the Monte Carlo type. Other examples are presented throughout the text. We say a Monte Carlo (Las Vegas) algorithm has *failed* if it does not give a correct answer (terminate within a specified amount of time).

1.4.3 Identifying the Repeated Element

Consider an array $a[\ ]$ of n numbers that has $\frac{n}{2}$ distinct elements and $\frac{n}{2}$ copies of another element. The problem is to identify the repeated element.

Any deterministic algorithm for solving this problem will need at least $\frac{n}{2} + 2$ time steps in the worst case. This fact can be argued as follows: Consider an adversary who has perfect knowledge about the algorithm used and who is in charge of selecting the input for the algorithm. Such an adversary can make sure that the first $\frac{n}{2} + 1$ elements examined by the algorithm are all distinct. Even after having looked at $\frac{n}{2} + 1$ elements, the algorithm will not be in a position to infer the repeated element. It will have to examine at least $\frac{n}{2} + 2$ elements and hence take at least $\frac{n}{2} + 2$ time steps.

In contrast there is a simple and elegant randomized Las Vegas algorithm that takes only $\widetilde{O}(\log n)$ time. It randomly picks two array elements and checks whether they come from two different cells and have the same value. If they do, the repeated element has been found. If not, this basic step of sampling is repeated as many times as it takes to identify the repeated element.

In this algorithm, the sampling performed is with repetitions; that is, the first and second elements are randomly picked from out of the n elements (each element being equally likely to be picked). Thus there is a probability (equal to $\frac{1}{n}$) that the same array element is picked each time. If we just check for the equality of the two elements picked, our answer might be incorrect (in case the algorithm picked the same array index each time). Therefore, it is essential to make sure that the two array indices picked are different and the two array cells contain the same value.

This algorithm is given in Algorithm 1.27. The algorithm returns the array index of one of the copies of the repeated element. Now we prove that the run time of the above algorithm is $\widetilde{O}(\log n)$. Any iteration of the **while** loop will be successful in identifying the repeated number if i is any one the $\frac{n}{2}$ array indices corresponding to the repeated element and j is any one of the same $\frac{n}{2}$ indices other than i. In other words, the probability that the algorithm quits in any given iteration of the **while** loop is $P = \frac{n/2(n/2-1)}{n^2}$, which is $\geq \frac{1}{5}$ for all $n \geq 10$. This implies that the probability that the algorithm does not quit in a given iteration is $< \frac{4}{5}$.

```
RepeatedElement($a, n$)
// Finds the repeated element from $a[1:n]$.
{
    while (true) do
    {
        $i$ := Random() mod $n + 1$; $j$ := Random() mod $n + 1$;
        // $i$ and $j$ are random numbers in the range $[1, n]$.
        if (($i \neq j$) and ($a[i] = a[j]$)) then return $i$;
    }
}
```

Algorithm 1.27 Identifying the repeated array number

Therefore, the probability that the algorithm does not quit in 10 iterations is $< \left(\frac{4}{5}\right)^{10} < .1074$. So, Algorithm 1.27 will terminate in 10 iterations or less with probability $\geq .8926$. The probability that the algorithm does not terminate in 100 iterations is $< \left(\frac{4}{5}\right)^{100} < 2.04 * 10^{-10}$. That is, almost certainly the algorithm will quit in 100 iterations or less. If n equals $2 * 10^6$, for example, any deterministic algorithm will have to spend at least one million time steps, as opposed to the 100 iterations of Algorithm 1.27!

In general, the probability that the algorithm does not quit in the first $c\alpha \log n$ (c is a constant to be fixed) iterations is

$$< (4/5)^{c\alpha \log n} \;=\; n^{-c\alpha \log \,(5/4)}$$

which will be $< n^{-\alpha}$ if we pick $c \geq \frac{1}{\log \,(5/4)}$.

Thus the algorithm terminates in $\frac{1}{\log \,(5/4)} \alpha \log n$ iterations or less with probability $\geq 1 - n^{-\alpha}$. Since each iteration of the **while** loop takes $O(1)$ time, the run time of the algorithm is $\widetilde{O}(\log n)$.

Note that this algorithm, if it terminates, will always output the correct answer and hence is of the Las Vegas type. The above analysis shows that the algorithm will terminate quickly with high probability.

The same problem of inferring the repeated element can be solved using many deterministic algorithms. For example, sorting the array is one way. But sorting takes $\Omega(n \log n)$ time (proved in Chapter 10). An alternative is to partition the array into $\lceil \frac{n}{3} \rceil$ parts, where each part (possibly except for one part) has three array elements, and to search the individual parts for the repeated element. At least one of the parts will have two copies of the repeated element. (Prove this!) The run time of this algorithm is $\Theta(n)$.

1.4.4 Primality Testing

Any integer greater than one is said to be a *prime* if its only divisors are 1 and the integer itself. By convention, we take 1 to be a nonprime. Then $2, 3, 5, 7, 11$, and 13 are the first six primes. Given an integer n, the problem of deciding whether n is a prime is known as *primality testing*. It has a number of applications including cryptology.

If a number n is composite (i.e., nonprime), it must have a divisor $\leq \lfloor\sqrt{n}\rfloor$. This observation leads to the following simple algorithm for primality testing: Consider each number ℓ in the interval $[2, \lfloor\sqrt{n}\rfloor]$ and check whether ℓ divides n. If none of these numbers divides n, then n is prime; otherwise it is composite.

Assuming that it takes $\Theta(1)$ time to determine whether one integer divides another, the naive primality testing algorithm has a run time of $O(\sqrt{n})$. The input size for this problem is $\lceil(\log n + 1)\rceil$, since n can be represented in binary form with these many bits. Thus the run time of this simple algorithm is exponential in the input size (notice that $\sqrt{n} = 2^{\frac{1}{2}\log n}$).

We can devise a Monte Carlo randomized algorithm for primality testing that runs in time $O((\log n)^2)$. The output of this algorithm is correct with high probability. If the input is prime, the algorithm never gives an incorrect answer. However, if the input number is composite (i.e., nonprime), then there is a small probability that the answer may be incorrect. Algorithms of this kind are said to have *one-sided error*.

Before presenting further details, we list two theorems from number theory that will serve as the backbone of the algorithm. The proofs of these theorems can be found in the references supplied at the end of this chapter.

Theorem 1.6 [Fermat] If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for any integer $a < n$. □

Theorem 1.7 The equation $x^2 \equiv 1 \pmod{n}$ has exactly two solutions, namely 1 and $n-1$, if n is prime. □

Corollary 1.1 If the equation $x^2 \equiv 1 \pmod{n}$ has roots other than 1 and $n-1$, then n is composite. □

Note: Any integer x which is neither 1 nor $n-1$ but which satisfies $x^2 \equiv 1 \pmod{n}$ is said to be a *nontrivial square root* of 1 modulo n.

Fermat's theorem suggests the following algorithm for primality testing: Randomly choose an $a < n$ and check whether $a^{n-1} \equiv 1 \pmod{n}$ (call this

Fermat's equation). If Fermat's equation is not satisfied, n is composite. If the equation is satisfied, we try some more random a's. If on each a tried, Fermat's equation is satisfied, we output "n is prime"; otherwise we output "n is composite." In order to compute a^{n-1} **mod** n, we could employ Exponentiate (Algorithm 1.16) with some minor modifications. The resultant primality testing algorithm is given as Algorithm 1.28. Here *large* is a number sufficiently *large* that ensures a probability of correctness of $\geq 1 - n^{-\alpha}$.

```
Prime0($n, \alpha$)
// Returns true if $n$ is a prime and false otherwise.
// $\alpha$ is the probability parameter.
{
    $q := n - 1$;
    for $i := 1$ to $large$ do // Specify $large$.
    {
        $m := q$; $y := 1$;
        $a :=$ Random() mod $q + 1$;
        // Choose a random number in the range $[1, n-1]$.
        $z := a$;
        // Compute $a^{n-1}$ mod $n$.
        while ($m > 0$) do
        {
            while ($m$ mod $2 = 0$) do
            {
                $z := z^2$ mod $n$; $m := \lfloor m/2 \rfloor$;
            }
            $m := m - 1$; $y := (y * z)$ mod $n$;
        }
        if ($y \neq 1$) then return false;
        // If $a^{n-1}$ mod $n$ is not 1, $n$ is not a prime.
    }
    return true;
}
```

Algorithm 1.28 Primality testing: first attempt

If the input is prime, Algorithm 1.28 will never output an incorrect answer. If n is composite, will Fermat's equation never be satisfied for any a less than n and greater than one? If so, the above algorithm has to examine just one a before coming up with the correct answer. Unfortunately, the answer to this question is no. Even if n is composite, Fermat's equation may be satisfied depending on the a chosen.

Is it the case that for every n (that is composite) there will be some nonzero constant fraction of a's less than n that will not satisfy Fermat's equation? If the answer is yes and if the above algorithm tries a sufficiently large number of a's, there is a high probability that at least one a violating Fermat's equation will be found and hence the correct answer be output. Here again, the answer is no. There are composite numbers (known as Carmichael numbers) for which every a that is less than and relatively prime to n will satisfy Fermat's equation. (The number of a's that do not satisfy Fermat's equation need not be a constant fraction.) The numbers 561 and 1105 are examples of Carmichael numbers.

Fortunately, a slight modification of the above algorithm takes care of these problems. The modified primality testing algorithm (also known as Miller-Rabin's algorithm) is the same as Prime0 (Algorithm 1.28) except that within the body of Prime0, we also look for nontrivial square roots of n. The modified version is given in Algorithm 1.29. We assume that n is odd.

Miller-Rabin's algorithm will never give an incorrect answer if the input is prime, since Fermat's equation will always be satisfied and no nontrivial square root of 1 modulo n can be found. If n is composite, the above algorithm will detect the compositeness of n if the randomly chosen a either leads to the discovery of a nontrivial square root of 1 or violates Fermat's equation. Call any such a a *witness* to the compositeness of n. What is the probability that a randomly chosen a will be a witness to the compositeness of n? This question is answered by the following theorem (the proof can be found in the references at the end of this chapter).

Theorem 1.8 There are at least $\frac{n-1}{2}$ witnesses to the compositeness of n if n is composite and odd. □

Assume that n is composite (since if n is prime, the algorithm will always be correct). The probability that a randomly chosen a will be a witness is $\geq \frac{n-1}{2n}$, which is very nearly equal to $\frac{1}{2}$. This means that a randomly chosen a will fail to be a witness with probability $\leq \frac{1}{2}$.

Therefore, the probability that none of the first $\alpha \log n$ a's chosen is a witness is $\leq \left(\frac{1}{2}\right)^{\alpha \log n} = n^{-\alpha}$. In other words, the algorithm Prime will give an incorrect answer with only probability $\leq n^{-\alpha}$.

The run time of the outermost **while** loop is nearly the same as that of Exponentiate (Algorithm 1.16) and equal to $O(\log n)$. Since this **while** loop is executed $O(\log n)$ times, the run time of the whole algorithm is $O(\log^2 n)$.

5. Given a 2-sided coin. Using this coin, how will you simulate an n-sided coin

   (a) when n is a power of 2?.

   (b) when n is not a power of 2?.

6. Compute the run time analysis of the Las Vegas algorithm given in Algorithm 1.30 and express it using the $\tilde{O}()$ notation.

```
LasVegas()
{
    while (true) do
    {
        $i$ := Random() mod 2;
        if ($i \geq 1$) then return;
    }
}
```

Algorithm 1.30 A Las Vegas algorithm

7. There are $\sqrt{n}$ copies of an element in the array c. Every other element of c occurs exactly once. If the algorithm RepeatedElement is used to identify the repeated element of c, will the run time still be $\tilde{O}(\log n)$? If so, why? If not, what is the new run time?

8. What is the minimum number of times that an element should be repeated in an array (the other elements of the array occurring exactly once) so that it can be found using RepeatedElement in $\tilde{O}(\log n)$ time?

9. An array a has $\frac{n}{4}$ copies of a particular unknown element x. Every other element in a has at most $\frac{n}{8}$ copies. Present an $O(\log n)$ time Monte Carlo algorithm to identify x. The answer should be correct with high probability. Can you develop an $\tilde{O}(\log n)$ time Las Vegas algorithm for the same problem?

10. Consider the naive Monte Carlo algorithm for primality testing presented in Algorithm 1.31. Here Power(x, y) computes x^y. What should be the value of t for the algorithm's output to be correct with high probability?

11. Let $\mathcal{A}$ be a Monte Carlo algorithm that solves a decision problem π in time T. The output of $\mathcal{A}$ is correct with probability $\geq \frac{1}{2}$. Show how

```
Prime1($n$)
{
    // Specify $t$.
    for $i := 1$ to $t$ do
    {
        $m :=$ Power($n$, 0.5);
        $j :=$ Random() mod $m + 2$;
        if (($n$ mod $j$) $= 0$) then return false;
        // If $j$ divides $n$, $n$ is not prime.
    }
    return true;
}
```

Algorithm 1.31 Another primality testing algorithm

you can modify $\mathcal{A}$ so that its answer is correct with high probability. The modified version can take $O(T \log n)$ time.

12. In general a Las Vegas algorithm is preferable to a Monte Carlo algorithm, since the answer given by the former is guaranteed to be correct. There may be critical situations in which even a very small probability of an incorrect answer is unacceptable. Say there is a Monte Carlo algorithm for solving a problem π in T_1 time units whose output is correct with probability $\geq \frac{1}{2}$. Also assume that there is another algorithm that can check whether a given answer is valid for π in T_2 time units. Show how you use these two algorithms to arrive at a Las Vegas algorithm for solving π in time $\widetilde{O}((T_1 + T_2) \log n)$.

13. The problem considered here is that of searching for an element x in an array $a[1 : n]$. Algorithm 1.17 gives a deterministic $\Theta(n)$ time algorithm for this problem. Show that any deterministic algorithm will have to take $\Omega(n)$ time in the worst case for this problem. In contrast a randomized Las Vegas algorithm that searches for x is given in Algorithm 1.32. This algorithm assumes that x is in $a[\,]$. What is the $\widetilde{O}()$ run time of this algorithm?

```
Algorithm RSearch($a, x, n$)
// Searches for $x$ in $a[1 : n]$. Assume that $x$ is in $a[\,]$.
{
    while (true) do
    {
        $i :=$ Random() mod $n + 1$;
        // $i$ is random in the range $[1, n]$.
        if ($a[i] = x$) then return $i$;
    }
}
```

Algorithm 1.32 Randomized search

1.5 REFERENCES AND READINGS

For a more detailed discussion of performance analysis and measurement, see *Software Development in Pascal*, Third Edition, by S. Sahni, NSPAN Printing and Publishing, 1993.

For a discussion on mathematical tools for analysis see *Concrete Mathematics: A Foundation for Computer Science*, by R. L. Graham, D. E. Knuth, and O. Patashnik, Addison-Wesley, 1989.

More details about the primality testing algorithm can be found in *Introduction to Algorithms*, by T. H. Cormen, C. E. Leiserson, and R. L. Rivest, MIT Press, 1990.

An excellent introductory text on probability theory is *Probability and Random Processes*, by G. R. Grimmet and D. R. Stirzaker, Oxford University Press, 1988. A proof of Lemma 1.1 can be found in this book. For a proof of Lemma 1.2 see *Queueing Systems,* Vol. I, by L. Kleinrock, John Wiley & Sons, 1975.

A formal treatment of randomized algorithms and several examples can be found in "Derivation of randomized algorithms for sorting and selection," by S. Rajasekaran and J. H. Reif, in *Parallel Algorithm Derivation and Program Transformation*, edited by R. Paige, J. H. Reif, and R. Wachter, Kluwer Academic Publishers, 1993, pp. 187–205. For more on randomized algorithms see *Randomized Algorithms* by R. Motwani and P. Raghavan, Cambridge University Press, 1995.